Aalto University
School of Electrical Engineering
Department of Electrical Engineering and Automation

Teemu Alonen

Inference with a neural network in digital signal processing under hard real-time constraints

Master's Thesis
Espoo, 31.12.2019

Supervisor: Prof. Themistoklis Charalambous
Advisors: Prof. Risto Wichman
PhD Jean-Luc Olives
MSc Marko Hassinen

Aalto University
School of Electrical Engineering
Department of Electrical Engineering and Automation

ABSTRACT OF
MASTER'S THESIS

Author:	Teemu Alonen		
Title:	Inference with a neural network in digital signal processing under hard real-time constraints		
Date:	31.12.2019	**Pages:**	viii + 75
Professorship:	Automation Engineering	**Code:**	ELEC0007
Supervisor:	Prof. Themistoklis Charalambous		
Advisors:	Prof. Risto Wichman PhD Jean-Luc Olives MSc Marko Hassinen		

The main objective of this thesis is to investigate how neural network inference can be efficiently implemented on a digital signal processor under hard real-time constraints from the execution speed point of view. Theories on digital signal processors and software optimization as well as neural networks are discussed. A neural network model for the specific use case is designed and a digital signal processor implementation is created based on the neural network model.

A neural network model for the use case is created based on the data from the Matlab simulation model. The neural network model is trained and validated using the Python programming language with the Keras package. The neural network model is implemented on the CEVA-XC4500 digital signal processor. The digital signal processor implementation is written in C++ language with the processor specific vector-processing intrinsics. The neural network model is evaluated based on the model accuracy, precision, recall and f1-score. The model performance is compared to the conventional use case implementation by calculating 3GPP specified metrics of misdetection probability, false alarm rate and bit error rate. The execution speed of the digital signal processor implementation is evaluated with the CEVA integrated development environment profiling tool and also with the Lauterbach PowerTrace profiling module attached to the real base station product.

Through this thesis, an optimized CEVA-XC4500 digital signal processor implementation was created for the specific neural network architecture. The optimized implementation showed to consume 88 percent less cycles than the conventional implementation. Also, the neural network model performance fulfills the 3GPP specification requirements.

Keywords:	neural networks, machine learning, digital signal processors, digital signal processing, 5G
Language:	English

Aalto-yliopisto
Sähkötekniikan korkeakoulu
Automaatio- ja systeemitekniikan laitos

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä:	Teemu Alonen		
Työn nimi:	Neuroverkon inferenssi digitaalisessa signaalikäsittelyssä kovien reaaliaikavaatimusten alaisuudessa		
Päiväys:	31.12.2019	**Sivumäärä:**	viii + 75
Professuuri:	Automaatio- ja systeemitekniikka	**Koodi:**	ELEC0007
Valvoja:	Prof. Themistoklis Charalambous		
Ohjaajat:	Prof. Risto Wichman TkT Jean-Luc Olives FM Marko Hassinen		

Tämän diplomityön tarkoituksena on tutkia miten neuroverkon inferenssi voidaan toteuttaa tehokkaasti digitaalisella signaaliprosessorilla suoritusnopeuden näkökulmasta, kun sovelluksella on kovat reaaliaikavaatimukset. Työssä käsitellään teoriaa digitaalisista signaaliprosessoreista, ohjelmistojen optimoinnista ja neuroverkoista. Työssä kehitetään neuroverkkomalli tiettyyn käyttötapaukseen, ja mallin pohjalta luodaan toteutus digitaaliselle signaaliprosessorille.

Neuroverkkomalli luodaan Matlab-simulointimallin avulla kerätystä datasta. Neuroverkkomalli opetetaan ja varmennetaan Python-ohjelmointikiellellä ja Keras-paketilla. Neuroverkkomalli toteutetaan CEVA-XC4500 digitaaliselle signaaliprosessorille. Digitaalisen signaaliprosessorin toteutus kirjoitetaan C++-ohjelmointikielellä ja prosessorikohtaisilla vektorilaskentaoperaatioilla. Neuroverkkomalli varmennetaan mallin tarkkuuden, precision-arvon, recall-arvon ja f1-arvon perusteella. Mallin suorituskykyä verrataan käyttötapauksen tavanomaiseen toteutukseen laskemalla 3GPP-spesifikaation mukaiset mittarit virhehavaintotodennäköisyys, väärien hälytysten lukumäärä ja bittivirhemäärä. Suoritusnopeus määritetään sekä CEVA ohjelmointiympäristön profilointityökalulla että tukiasematuotteeseen kytketyllä Lauterbach PowerTrace-yksiköllä.

Työn tuloksena luotiin optimoitu CEVA-XC4500 digitaalinen signaaliprosessoritoteutus valitulle neuroverkkoarkkitehtuurille. Optimoitu toteutus kulutti 88% vähemmän laskentasyklejä kuin tavanomainen toteutus. Neuroverkkomalli täytti 3GPP-spesifikaation mukaiset vaatimukset.

Asiasanat:	neuroverkot, koneoppiminen, digitaalinen signaalin käsittely, digitaalinen signaaliprosessori, 5G
Kieli:	Englanti

Preface

This thesis was written in 2019 at Nokia in Espoo headquartes, Finland.

I would like to thank Nokia for giving me this opportunity and such an interesting and motivating topic for this thesis. I want to give a special thanks to my advisors Jean-Luc Olives and Marko Hassinen as well as other colleagues in the L1 machine learning and the L1 development teams for supporting my work and brainstorming new ideas. I also want to thank my supervising professors Themistoklis Charalambous and Risto Wichman for their great support.

I am also thankful for my family and friends, who have always supported me through my whole academic journey.

Espoo, 31.12.2019

Teemu Alonen

Symbols and abbreviations

Symbols

t	time [s]
c	cycle count
f	frequency [Hz]
I	total number of instructions
a	accumulator
w	neural network weight coefficient
W	neural network weight coefficient matrix
x	neural network input value
X	neural network input vector
b	neural network bias coefficient
y	actual neural network output value
Y	actual neural network output vector
d	desired neural network output value
v	induced local field of a neuron
η	learning rate
e	error value
C	cost function
λ	regularization factor

Abbreviations

AD	Analog-to-digital
AGA	Adaptive gradient algorithm
ANN	Artificial neural network
ASIC	Application-specific integrated circuit
BER	Bit error rate
CISA	Configurable instruction set architecture
CPI	Clock cycles per instruction
CPU	Central processing unit
DAAU	Data address and arithmetic unit
DMA	Direct memory access
DSP	Digital signal processor
DTX	Discontinuous transmission
FAR	False alarm rate
FFT	Fast fourier transform
GCU	General computation unit
HLL	High level language
MAC	Multiply-accumulate
MDP	Misdetection propability
MPN	McCulloch-Pitts Neuron
MLP	Multilayer perceptron
NN	Neural network
PCU	Program control unit
PSU	Power scaling unit
ReLU	Rectified linear unit
RISC	Reduced instruction set computer
RMSP	Root mean square propagation
RNN	Recurrent neural network
SIMD	Single instruction multiple data
SNR	Signal-to-noise ratio
SOP	Sum of products
VCU	Vector computation unit
VLIW	Very long instruction word
VRF	Vector register file

Contents

Abstract ii

Tiivistelmä iii

Preface iv

Symbols and abbreviations v

Contents vii

1 Introduction 1
- 1.1 Background . . . 1
- 1.2 Research Problem . . . 1
- 1.3 Methods and Materials . . . 2
- 1.4 Outline of the thesis . . . 3

2 Digital signal processor 4
- 2.1 Overview . . . 4
- 2.2 Definitions for real-time . . . 5
- 2.3 DSP architecture . . . 7
  - 2.3.1 Performance metrics . . . 7
  - 2.3.2 Representation of numbers . . . 8
  - 2.3.3 Data path . . . 10
  - 2.3.4 Memory architecture . . . 11
- 2.4 General optimization methods . . . 14
  - 2.4.1 Optimization algorithms . . . 15
  - 2.4.2 Effective use of DSP architecture . . . 17
  - 2.4.3 Compiler optimization . . . 17
- 2.5 CEVA-XC4500 DSP . . . 21

3 Artificial neural networks **25**
3.1 Introduction . . . 25
3.1.1 McCulloch-Pitts neuron . . . 27
3.1.2 Rosenblatt's perceptron . . . 28
3.2 Multilayer perceptrons . . . 33
3.3 Training procedure . . . 34
3.3.1 Cost function . . . 36
3.3.2 Backpropagation of errors . . . 37
3.3.3 Optimization algorithms . . . 38
3.3.4 Regularization . . . 40

4 Implementation **41**
4.1 Use case . . . 41
4.2 Neural network model . . . 44
4.2.1 Data . . . 46
4.2.2 Model architecture and training . . . 46
4.2.3 Activation functions . . . 49
4.3 DSP implementation . . . 49
4.3.1 Fixed point format . . . 50
4.3.2 Induced local field of a neuron . . . 53
4.3.3 Bias . . . 56
4.3.4 Activation functions . . . 57

5 Evaluation **60**
5.1 Neural network model . . . 60
5.1.1 Evaluation metrics . . . 60
5.1.2 Evaluation . . . 61
5.2 DSP implementations . . . 63
5.2.1 Evaluation metrics . . . 64
5.2.2 Evaluation . . . 65
5.3 Evaluation summary . . . 68

6 Conclusions **69**

References **71**

Chapter 1

Introduction

1.1 Background

This thesis is examining CEVA-XC4500 digital signal processor (DSP), which is part of the Nokia ReefShark chipset. ReefShark chipset is Nokia's in-house system-on-chip (SOC) module developed for Nokia baseband products. ReefShark chipset is based on 3GPP specifications for 4G and 5G New Radio (NR), and it is delivered as plug-in unit for the commercially available Nokia AirScale baseband module. AirScale module is based on the idea of software-defined system modules, and it supports all radio technologies from 2G to 5G, and all network architectures from distributed radio-access networks (RAN) to centraliced RAN, including also cloud RAN capability.

1.2 Research Problem

Mobile network requirements are getting more demanding in the 2020s due to massively increasing data rates. 5G is planned to meet new requirements, but it brings more engineering challenges due to aggregated data rates, higher edge data rates and peak rates, increasing amount of supported simultaneous user equipment, tighter latency requirements and unknown channel models [1]. Because these new features challenge conventional communication theories, machine learning is one of the proposed solutions to solve them. Machine learning has been widely applied to the upper layer of wireless communication systems for various purposes, and it is increasingly recognized also in the physical layer development [2] [3].

Most of the high-speed processing in the physical layer is done by digital signal processors or specialized hardware units. Machine learning could be optimally processed using special ML processing chip, but if none is available, ML-based

processing needs to be done on a digital signal processor. Digital signal processors are optimized for multiply-accumulate (MAC) operations, as many of the modern digital signal processing algorithms, such as convolution, are based on those operations. There is fundamental similarity to feedforward neural networks, whose forward propagation is mostly processed by multiplications and additions.

Basically, the research problem is defined as follows: how a machine learning algorithm can be efficiently implemented on a digital signal processing embedded system for real-time applications. This question includes identifying methods, and designing the machine learning algorithm from the DSP architecture and operations point-of-view. In addition, an important research question is to cover what are the advantages, limitations and concerns when utilizing digital signal processor architecture and operations for machine learning purposes.

1.3 Methods and Materials

Methods of developing execution speed-optimised software for digital signal processors is reviewed by using available literature. Theory behind neural networks, especially in the context of physical layer software development, is presented as a basis for this research.

Since this thesis work is closely related to Nokia R&D project to utilize machine learning in physical layer software, specific information about the use case algorithm is deliberately concealed. All the relevant information including input and output data dimensions and precision regarding to the machine learning algorithm are shared, but the specific use case description and absolute performance metrics are not presented.

The neural network inference algorithm is implemented in C++ programming language with CEVA-provided header files containing CEVA XC-specific macros for vectorized processing. The implementation is compiled with CEVA-provided compiler. The implementation is run on a CEVA-XC4500 digital signal processor. Real-time profiling data of the implementation performance is collected using the Lauterbach PowerTrace module attached to the digital signal processor. Data for the neural network model training and validation is collected using Nokia's 5G simulator, which is built by Matlab software. The neural network model itself is constructed and trained using python programming language with Keras and Tensorflow modules.

1.4 Outline of the thesis

The aim of this thesis is to apply machine learning techniques in physical layer software development by implementing computationally speed-efficient neural network inference for CEVA-XC4500 digital signal processor. The purpose is also to apply the implementation for physical layer control bit decoding use case, and compare its performance against the implementation based on the stochastic mathematical models. The thesis work is divided into 6 chapters. Chapter 2 provides background related to digital signal processors in general. It also describes most commonly used methods to optimize digital-signal processor applications, and defines the concept of real-time. Chapter 3 covers a machine learning related algorithm called neural networks. The neural network model itself, network training and network inference are discussed. After introduction and theoretical recap, chapter 4 describes how neural network architecture and digital signal processor implementation are tied together on experimental level. Different CEVA-XC4500 digital signal processor related optimization methods are explained, and how neural network model is best fitted to both CEVA-XC4500 specifications and application purposes. Chapter 5 presents evaluation and results how implemented neural network inference compares against stochastic model based implementation. Chapter 6 includes the review of the implementation and discusses the future improvements and requirements related to machine learning in physical layer development.

Chapter 2

Digital signal processor

2.1 Overview

Digital signal processing refers to a set of mathematical operations to digitally represent signals [4]. The goal of the digital signal processing is to determine specific information content by transforming, enhancing and modifying the signal. Digital signal processing involves the processing of analog signals that are converted and represented digitally by sequence of numbers. The term 'digital' refers to this numerical representation, which also implies quantization of some of the signal properties. [4]. The term 'signal' refers to a variable parameter, which is treated as information as it flows through an electronic circuit [4]. The signal is essentially a voltage that varies within some range of values [4]. The term 'processing' relates to the processing of data using software based applications [4].

General-purpose processors (GPPs) are designed to provide broad functionality for a wide variety of different kind of applications [4]. From the performance point of view, the goal of GPPs is to maximize performance over a broad range of applications. Specialized processors instead, are designed to take maximum advantage of the limited functionality required by their special applications. [4]. Digital signal processor (DSP) is a processor specialized for digital signal processing. Its hardware is shaped by the digital signal processing algorithms, and thus it is specialized to perform them inexpensively and efficiently. Inexpensive and efficient refers to different DSP performance metrics, which are usually the processing time required to accomplish a defined task, memory usage and energy consumption. [5]. In the literature, the acronym DSP can refer to both 'digital signal processing' and 'digital signal processor'. In this thesis, DSP refers to the 'digital signal processor'.

The strict role between GPP and DSP is blurring, because many so-called GPPs have some DSP functionalities, and many DSPs have traditional general-purpose processing functionalities on board [6]. Strictly speaking, any processor

that operates on digitally presented signals can be called DSP. In practise, however, DSP refers to a processor specifically designed for digital signal processing [5]. In this thesis, DSP refers to the latter. Distinguishing between DSPs and GPPs by application is perhaps not the best way forward. The main difference between those two lies inside the devices themselves, in the internal chip architecture [6].

This chapter introduces the basics of DSPs from the application optimization point of view. Especially, optimization from application speed point of view is discussed, as the purpose of this thesis is to produce as fast neural network real-time inference as possible. The term real-time is defined and discussed in section 2.2. DSP architecture overview is discussed in section 2.3, and DSP memory architecture is discussed more in detail in section 2.3.4. Section 2.3.3 explains the meaning of data path, and why it is important aspect in DSP architecture. Section 2.4 discusses some of the most common methods to optimize DSP applications, and how to utilize DSP architecture efficiently. Lastly, section 2.5 discusses in detail about the architecture of the CEVA-XC4500 DSP, as the implementation of the real-time neural network inference is implemented on that chip.

2.2 Definitions for real-time

In order to define the term real-time, it is necessary to define a system. Based on the definition in [7], a system is a mapping of a set of inputs into a set of outputs. When internal details are out of interest, the mapping function can be considered as a black box. Different system definitions may include some other requirements for a system, like system must have purpose [8], but for practical engineering definition of real-time, input-output mapping is the key concept [7]. Every real-world entity can be modeled as a system [7]. In computing systems, like DSPs, the inputs and outputs represent digital or analog data. In embedded systems, inputs may be associated with sensors, and outputs may be connected to the actuators. Figure 2.1 represents the general model of a system with input-output mapping.

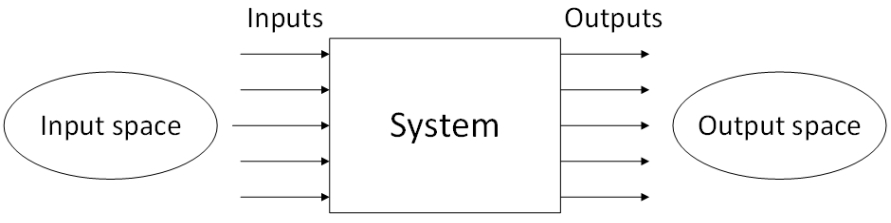

Figure 2.1: A general system with inputs and outputs [7].

Figure 2.2 represents a classical model of a real-time system. Instead of just using digital or analog inputs and outputs, system excitations are considered as

a sequence of jobs to be scheduled. Also, the performance of the jobs can be predicted [7]. It is still notable that this real-time model ignores the fact that the system inputs and the controlled hardware may be very complex, but it it still good representation of a real-time system.

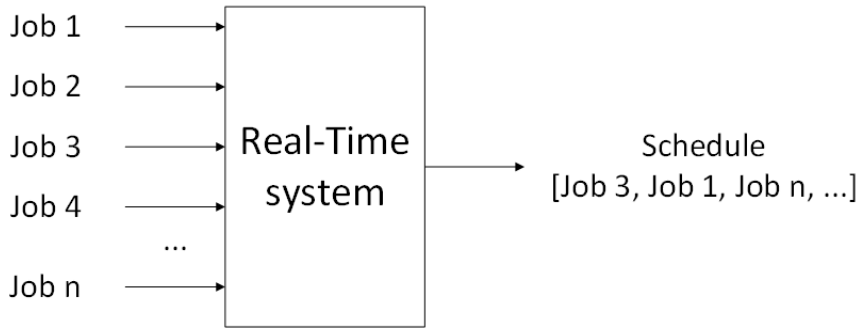

Figure 2.2: A real-time system as a sequence of jobs [7].

Both in general system model and in real-time system model, there exists a delay between presentation of inputs and appearance of the output [7]. This is one of the key reasons why definition of real-time is not just instantaneous response. This delay is called as a response time of the system [7]. The response time of the system is defined as "the time between the presentation of a set of inputs to a system and the realization of the required behavior, including the availability of all associated outputs" [7]. System response-time differs from one application to another, and the requirements for response-time solely depends on the characteristics and the purpose of the system. However, the response-time requirement defines the term real-time system. A real-time system must satisfy bounded response-time constraints or risk severe consequences, including failure [7]. A failure in a system means that the system cannot satisfy one or more of the requirements defined in the system requirements specification [7]. Because of this definition of failure, system operation criteria and timing constraints must be defined in order to discuss about a real-time system. Because of existing timing criterion, the real-time system logical correctness is evaluated based on the correctness of the outputs and the fullfilled response time constraints. It is notable that a real-time system does not have to process data instantaneously, it just need to have response times that are satisfied under defined constrains.

One important aspect is also the fact that some applications may accept different amount of failures without catastrophic consequences. The affect of failure in a nuclear reactor cooling system response time has a very different consequence than flight ticket reservation system. Real-time systems are traditionally divided into three categories [7] based on the effect of missed real-time constraints; soft real-time systems, hard real-time systems and firm real-time systems. In a soft real-time

system the performance is degraded if the response time constraints are not met, but the system is not completely destroyed [7]. In a hard real-time system instead, even a single missed response-time constraint may lead to a complete system failure [7]. In a firm real-time system, a few missed response-time constraints will not lead to a system failure, but missing more than a few may destroy the system [7].

2.3 DSP architecture

Architectural choices vary between different DSP vendors, but some characteristics are common to all DSPs. Typically, the DSP hardware is designed to support fast arithmetic by utilizing large accumulators, implementing single cycle multiply-accumulate (MAC) instructions, and supporting pipelined and parallell computation and data movement [4]. Parallel data movement is typically enabled by having multiple-access memories, which allows the processor to load and store multiple operands simultaneously and even in parallel with an instruction execution [9]. High bandwidth memory subsystems enable constant flow of operands available. DSPs typically feature also hardware support for low overhead loop control, and specialized instructions and addressing modes that reduce the total number of instructions required to describe a typical DSP algorithm [6]. Different memory-addressing modes and program-flow controls speed the execution of repetitive operations [9]. Also, special on-chip peripherals or input-output interfaces are included, so that the processor can interface efficiently with other system components, such as analog-to-digital (AD) converters and memory [9].

2.3.1 Performance metrics

One of the key performance measures discussed throughout this thesis is the processing time t required to process an algorithm. As discussed in section 2.2, response-time constraints need to be defined in real-time systems in order to evaluate system performance. Instead of discussing about the absolute time the DSP system consumes, DSP cycle count is discussed instead. Cycle count means the amount of central processing unit (CPU) clock ticks some measured instant takes. If the DSP main clock runs on a constant frequency within algorithm execution, the absolute time consumed by the algorithm can be determined from the cycle count according the equation 2.1

$$t = \frac{c}{f} \tag{2.1}$$

where t is the absolute CPU time consumed, c is the total cycle count, and f is the (constant) CPU clock frequency during execution. Because processing time

can be estimated from the cycle count, the cycle count is discussed in this thesis as an indicator for processing speed.

The required processing time can also be estimated from the total amount of instructions the algorithm includes, without knowing the consumed cycle amount. In equation 2.2, I refers to the total amount of instruction in the algorithm, and CPI refers to the average number of cycles each instruction takes to execute, clock cycles per instruction [10].

$$t = \frac{I \times CPI}{f} \tag{2.2}$$

DSP performance can also be measured in DSP memory usage and power consumption. In some applications, those metrics might be as important, or even more important than processing speed. An ideal technique for measuring overall performance of the DSP system would yield data from on execution time, memory usage and power consumption. However, processing speed is commonly the primary measure of performance, with memory consumption and power usage as secondary considerations [9].

It is important to differentiate the DSP clock cycle count from instruction cycle count. Instruction cycle means processing of one CPU instruction. The instruction cycle consists of five different phases in a classic reduced instruction set computer (RISC) pipeline: fetch instruction, decode instruction, load operand, execute arithmetic function, and store the result [7]. These different instruction cycle phases may take different amount of CPU clock cycles depending on the instruction itself. In a DSP processor, one multiply-accumulate (MAC) instruction may take only one CPU clock cycle [9], but other instructions typically take multiple CPU clock cycles. DSP cycle count means the CPU clock cycle count, not the number of consumed instruction cycles.

2.3.2 Representation of numbers

Digital signal processing can be separated into fixed-point processing and floating-point processing. These designations refer to the format used to store and manipulate numeric representations of data, especially to the representation of decimal numbers. The difference between those two processing methods is in the content of stored information about the number. In a floating point format, the processor knows everything about the number. The processor knows how the number is stored, and what is the magnitude of it. In a fixed-point format, the scaling factor, or exponent, needs to be stored separately. [11]. It is the developer's responsibility to take care of the scaling factor. Figure 2.3 represents how floating-point and fixed-point values are stored on 32-bit format. In the figure, floating-point value is represented in IEEE-754 single-precision format. The letters indicate the parts of

the number stored in this floating-point format. Letter s indicates that the most significant bit is the sign bit. The e indicates 8 bits for the exponent. Finally, the m indicates 23-bit mantissa of the number. The key difference in the computation between floating-point values and fixed-point values is the responsibility and handling of the scaling factor. Hardware manages normalization and scaling of the numbers and exponents when processing floating point values, whereas in fixed point format, it is developer's responsibility. [11]. Modern DSPs offer both fixed- and floating-point arithmetic, even though traditionally DSP units supported only fixed-point arithmetic [12].

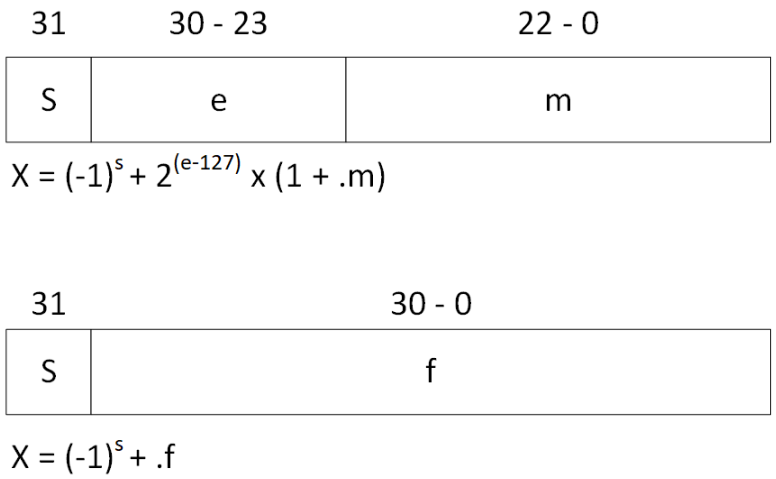

Figure 2.3: Single-precision 32-bit floating point value in IEEE-754 format on the top, typical 32-bit fixed-point value on the bottom [11].

Key consepts in the finite precision number representation are precision, resolution, accuracy and range. Precision refers to a maximum number of non-zero bits representable. In a fixed-point representation, precision equals to the word length. Resolution is the smallest non-zero magnitude representable. Range is the difference between the most negative and the most positive number representable. Accuracy refers to the magnitude of the maximum difference between a real value and it's representation. [13].

Floating-point representation has both advantages and disadvantages compared to fixed-point arithmetic. Floating-point arithmetic simplifies programming by making it easier to use higher level languages instead of assembly [12]. With floating-point values, the developer does not need to keep track the binary point position like with fixed-point values. The developer does not need to worry about proper scaling, data truncation and rounding. Floating-point arithmetic offers also greater dynamic range and precision [11]. Dynamic range is the range of numbers

that can be presented before an overflow occurs. Precision measures the number of bits to represent numbers. Precision can be used to estimate the impact of errors due to integer truncation and rounding [12].

Floating-point arithmetic has some disadvantages also. Some algorithms do not need floating-point scaling and range precision, and thus are better to be implemented on fixed-point values. Floating-point arithmetic is slower to process due to larger device size and more complex operations and more limited parallelism [12], which might also be a major performance drawback. Floating point operations hardware is typically larger, requiring more hardware space for the DSP. This also requires more power to operate. Added complexity also makes the hardware more expensive than pure fixed-point device [12]. Of course, the trade-off should be made regarding device cost compared to the software development cost on more demanding fixed-point arithmetic.

As discussed in section 2.4, optimization of the DSP application is usually trade-off between multiple metrics. Choosing between fixed-point and floating-point arithmetic is also trade-off between application speed and precision. This trade-off solely depends on the application requirements.

2.3.3 Data path

Data path refers to a set of functional units, such as multipliers, accumulators, registers and specialized units, which carry out all the arithmetic processing [14].

Multiplication is one of the key operations in digital-signal processing applications. Hence all DSPs have a multiplier that can multiply two data units in a single instruction cycle [14]. In some DSPs, the adder unit is separated from the multiplication unit, but in most of the DSPs, the adder is integrated with the multiplication unit. When adder and multiplier are integrated, they form single-cycle MAC-unit. If the units are separated, the result of the multiplication is first kept in a separate product result register before sending it to the adder for accumulation. This adds delay of at least one instruction to the processing [14]. The product of two n-bit fixed-point values will need $2n$ bits to store the result in order to avoid any accuracy lost. Most fixed-point multipliers produce a result that is twice the word-length of their operands [14], so the multiplier itself does not introduce any error. If the result of the multiplication is truncated, so that for example 32-bit result is truncated to 24-bits, some accuracy will be lost.

Pipelining is also utilized in some DSP multipliers. Pipelining is a technique that allows operations to overlap during program execution [6]. The task is split into multiple sub-tasks, which are overlapped. This increases the overall speed, even though there is delay between time the inputs are presented to the multiplier to the time that results are available. Single multiplication might have worse latency compared to non-pipelined multiplication, but long series of multiplications are

more effective. [14].

In addition to the multipliers, also accumulators are a fundamental part of digital-signal processing [14]. If there is only one accumulator available in the DSP architecture, and it is used as one of the source operands and also as the destination of the calculation, it can become the bottleneck of the processing. However, many DSPs offer more than one accumulator [14], and they are in many cases merged with the multiplication unit [6]. Accumulating two n-bit fixed-point values require $n+1$-bits for the resulting operand. Therefore, the size of the accumulator should be larger than the multiplier word by several bits, called guardian bits [14]. Guardian bits allow the accumulation of a number of results without overflow. Some DSPs that do not offer guard bits, allow scaling of the output register by shifting the value by a few bits. The scaling usually happens within a single instructions and is performed before adding the value to the accumulator. Guard bits are still more preferable, because they do not lost any precision. [14].

2.3.4 Memory architecture

The overall structure and the architecture of the memory is very important issue addressed in the design of the DSPs. Memory can be roughly divided into two types: internal memory and external memory. Internal memory referes to the DSP's on-chip memory [12], whereas external memory refers to peripheral off-chip memory [15]. Internal memory is much faster than off-chip external memory, therefore being the preferred storage for the processing data. Internal memory is sometimes interpret as sort of developer managed cache type of memory, as many DSPs do not have actual cache [12].

Most of the microprocessors are using memory design around the Von Neumann architecture [14], in which program instructions and program data are sharing the same memory space [6]. Figure 2.4 represents the Von Neumann based memory architecture. Because the memory space is shared, instructions and data are also accessed using the same buses, which makes the systems slower. Because of the shared bus, CPU needs to fetch both program instructions and data, before it can start processing itself. In Harvard architecture instead, which is used in most of the DSPs, program instructions and data are stored in separate and independent memory areas, and they are also accessed through separate buses [14]. Figure 2.5 represents the Harvard architecture with two separate banks of memory, which can be accessed in parallel. Because both instructions and data can be accessed simultaneously, speed advantage is gained compared to conventional Von Neumann based microprocessor architecture [6]. Also, the so called multi-port memories allows simultaneous memory access for data and instructions. Those memories have multiple independent sets of address and data lines, which allows multiple independent memory accesses [14].

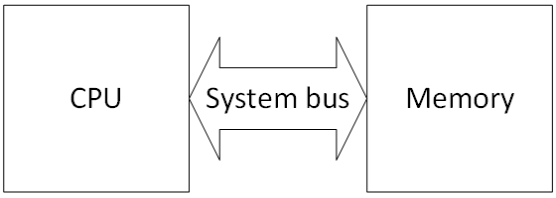

Figure 2.4: Von Neumann computer architecture without external input or output elements.

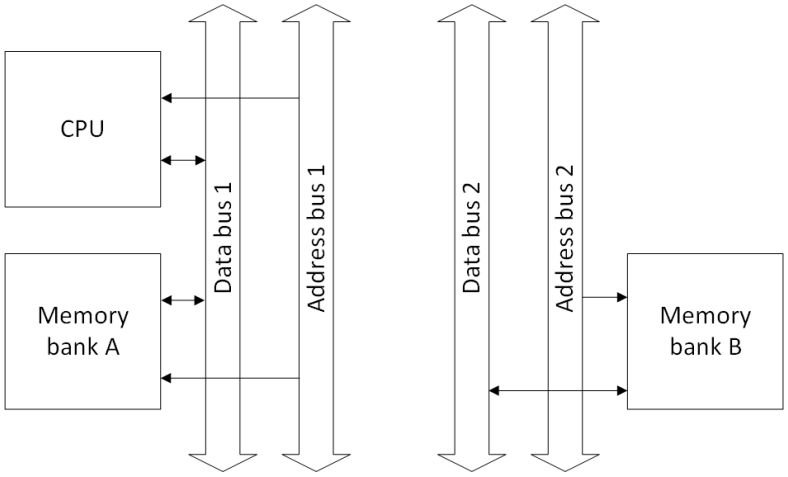

Figure 2.5: Harvard computer architecture.

There are some DSPs that implement more than two banks of memory, enabling even more independent memory accesses per instruction. However, this kind of multiple bus structure is expensive to be extended to external memory. Therefore, usually only one address and one data bus are available for off-chip external memory. [14]. Multiple memory access in one instruction can be achieved also with multiple access memories. In those, memory can be accessed in a fraction of an instruction cycle, thus allowing sequential access to be made on a single memory bus [14].

DSPs typically implement an architecture which is able to fetch multiple data and instruction within one cycle [4]. Most of the DSP architectures also have separate load and store instructions, which are the only ones allowed to access the memory. This way it is left to the compiler to schedule the rest of the work as how to utilize them [4]. If other instructions are allowed to access memory, barriers are created from the instruction level parallelism point of view, because this causes additional latency to the memory operations and data availability. Most DSPs vary in terms of addressing modes, access sizes and alignment restrictions on access to memory [4]. Some of the most common addressing modes in DSPs are as follows [4]:

- Register addressing. In this addressing mode, a register contains the operand. Because data processing between registers does not involve memory, this mode provides the fastest processing of data.
- Direct addressing. The address is part of the instruction itself, and the programmer specifies the address within the instruction. Direct access to the main memory is required, which results in slower processing.
- Register post increment or decrement addressing. The address pointer will take a register value and increment or decrement this value by a default step value.
- Segment plus offset addressing. The addrss pointer will take a register value added with a predefined offset from the instruction.
- Indexed addressing. The address pointer will take a register value and add an indexed register value.
- Modulo addressing. This addressing mode provides an automated means to support circular data buffers using hardware. This addressing mode remove the need to perform software based address boundary checks.
- Bit reversed addressing. Given the address of a particular element in the array, the DSP hardware automatically computes the address of the next element in the bit-reversed sequence.

Most of the DSP architectures are able to access memory at varied data widths [4]. Many DSP can access for example 8, 16 or 32 bit data operands. In addition to those generic data widths, many DSP architectures can often access 20 or 40-bit operands as well, because some operands result wider data resolution output operands than the input operands [4]. Very long instruction word (VLIW) is a common architecture for DSPs, meaning that compiler can break special, very long set of program instructions down into basic operations that can be performed in parallel [4]. This means that the DSP that has example 128-bit load-store bandwidth may be able to access 8 pieces of 32 bit data elements in parallel from the memory. This is very useful feature in situations where instruction level parallelism is high. By accessing multiple elements in parallel, and computing across them with vectorized single instruction multiple data (SIMD) instructions, the ratio of instructions in the program to computation performed is decreased, thereby yielding higher performance [4].

Program caches

A program cache is a small amount of memory for storing program instructions within the processor core [14]. Computations can be executed faster when the instruction is available at the processor core, without fetching it from the program memory. However, all of the DSPs does not have cache memory, because it causes determinism unpredictability in the calculations [12].

There are differences between processors how much the developer is able to control the cache memory usage. In some processors, the developer is able to lock the contents of the cache memory, or even disable its usage. This kind of manual control adds determinism, as it helps the developer to ensure that the programs will meet time constraints. [14].

If physical cache memory is not available in the processor, a similar type of idea can be utilized with internal and external memories. In a process called manual caching, the programmer can manually move some section of the code from slow external memory to the fast internal memory for execution. [14].

Direct memory access

Direct memory access (DMA) is the process of transferring data without the involvement of the processor itself [14]. Modern DSPs can often compute results faster that the memory system can supply new operands. The bottleneck is keeping the processor unit fed with data fast enough to prevent the system being idle waiting for new operands to be available. This situation is called as data starvation [12]. DMA is a solution to that problem. It is often used for transferring data between core and peripheral devices [14], such as external memory, which is very slow compared to the internal memory. External memory refers to a memory unit, which is located outside the chip. In DMA, a separate DMA controller is used for data transfer. The DMA controller is actually another CPU, who is only responsible of moving data around very quickly [12]. When the DMA controller is ready for a data transfer, it notifies the DSP, which in turn relinquishes its external memory bus control to the DMA controller. The DMA controller transfers the data independently from the DSP, and notifies the processors after completion of the transfer. [14]. DMA is most useful when transferring large block of data, as the setup and overhead time for the DMA transfer makes it faster just to use regular DSP control for smaller data blocks [12].

2.4 General optimization methods

Optimization is a procedure that seeks to maximize or minimize one or more performance indices without changing the meaning of the program output. In

the context of a real-time application, these performance indices typically include throughput, memory usage, external input and output bandwidths and power dissipation [12]. However, it is typically difficult or even impossible to optimize all of the performance indices at the same time. For example, the speed of the DSP algorithm is usually inversely proportional to the memory usage and power consumption of the algorithm, so that making the application faster requires more memory usage and more heat dissipation. The art of the optimization is knowing the different optimization options, understanding the trade-off between various performance indices, and without forgetting the overall goal of the application, creating the application based on those [12]. As many of the modern DSP applications are subject to real-time constraints, it is important to be aware of the general DSP optimization techniques. As discussed in the chapter 2.2, the definition of the real-time system is to be able to perform tasks in predetermined time intervals. As CPU power, memory size and power resources are valuable assets from the cost point of view, it is usually more cost-efficient trying to compress the application to use as little resources as possible. Therefore, the application should be as optimized as possible. Sometimes optimization may speed up the application by order of magnitudes [12]. Of course, DSPs differ from one to another, but due to limited special functionality requirements by the common signal processing applications, DSPs include some common key features to perform those tasks. Optimization of the DSP application is highly related to the efficient usage of DSP architecture, algorithms and the compiler [16]. Optimization methods represented in this chapter are common methods to improve the performance of a DSP application in terms of cycle count and memory usage.

2.4.1 Optimization algorithms

Algorithm optimization is the highest level of DSP software optimization. Before implementing and optimizing the code, one should try to optimize the algorithm itself and try to make it as efficient as possible [16]. It should be optimized especially from the DSP point of view, so that DSP architecture and compiler are taken into account while formulating the algorithm. Also, data size to be processed need to be minimized [16], as it generally decreases the amount of required processing. Choosing the right data structure for the right application will provide an efficient way of accessing data and therefore improve the performance [16]. Some data structures, like classes, might be harder for the compiler to optimize. Some common programming related code optimization techniques [12], which also help in compiler optimization, are:

- Code rearrangement. Changing the order of code execution might save time, if memory read is triggered little bit earlier than the operands are needed.

Some other code can be executed between the read and execution.

- Minimize branching. Branching is typically harmful from the pipelining point of view. As pipelining allows sequential instruction to be executed simultaneously, there should be knowledge about the instruction to be executed next. Branching brokes this determinism, as the forecoming path is unknown. As discussed in the chapter 2.5, vector predicates is a method to execute conditional operations without conditional branching.

- Elimination of recalculations. If it is possible to avoid calculating something again, for example moving code to the outside of the loop, both speed and code size are optimized.

- Combining equivalent constants and substituting operations. It is suitable to combine constant operations beforehand, so that execution time does not need to be used for it. Also, if two constants have equal values, they should be replaced by a single constant for memory saving purposes.

- Eliminating unused code and storage of unreferenced values. All of the unused code should be removed, as it consumes memory.

- Inlining or replacing function calls with program code. Inlining refers to a method in which the compiler replaces part of the code, for example function call, with the copy of the source code. This prevents the execution jumping from one code section to another, enabling pipelining possibilities, It can speed up the execution of the software by not having to perform function calls with the associated overhead. However, inlining increases code size. [12].

- Loop unrolling. Loop unrolling is a technique in which the body of a suitable loop is replaced with multiple copies of itself, and the control logic of the loop is updated accordingly [17]. Loop unrolling attempts to minimize the the cost of loop overhead, such as branching on the termination condition and updating counter variables [12]. The loop unrolling manually adds multiple copies of the code body, and ads updates and counter increments accordingly. This increases code size, but the speed performance is potentially improved. The performance benefit comes from the reduced loop overhead, because less iterations are performed, and code is potentially pipelined more [12].

- Loop invariant code motion. If a variable within a loop is not altered, the calculation should be performed outside of the loop body [12].

2.4.2 Effective use of DSP architecture

DSP is essentially an application-specific microprocessor. DSP was developed to include hardware architectures that allow the efficient execution of signal processing specific algorithms [12]. As discussed previously in this chapter, some of the specific architectural features of DSPs include special instructions, large accumulators, specialized loop checking and multiaccess memories [12]. Special hardware-based instructions speed up the instruction execution and large accumulators allow accumulating a large number of elements [12]. Multiaccess memories allows accessing two or more data elements in the same cycle, and special loop checking hardware performs much faster than software-based loop checking [12].

As discussed in section 2.3, many DSP applications are composed from a standard set of DSP building blocks, such as filters, FFT and convolutions [12]. What is common to these algorithms, they all perform series of multiplies and adds, which is commonly referred to as sum of products (SOP). One of the most common operation, which is encountered in all of these major DSP functions, is the multiply-accumulate (MAC) operation [12, 14]. MAC operation is represented in equation (2.3), in which w and x are operands to be first multiplied, and then accumulated to an accumulator a. MAC operation computes the product of two numbers, and adds that product to an accumulator. Many DSP related algorithms, like FFT, perform multiple of those operations within a tight loop. As shown in [12], the saving of calculating MAC as a hardware-dedicated operation is four cycles compared to the software or microcode based operation. The saving becomes more and more significant, when multiple of MAC operations are performed millions of times in an application.

$$a = a + (w \times x) \tag{2.3}$$

2.4.3 Compiler optimization

Compiler is a computer program that translates computer code written in one programming language into another programming language [12], usually into machine code. Compilers are used to translate the source code from higher level language to lower level language to create an executable program. Figure 2.6 represents the general architecture of a modern compiler [12]. The front end of the compiler reads in the source code, report errors and creates an intermediate representation of the source code [12]. The intermediate stage is the optimizer. The back end of the compiler generates the target code from the optimized intermediate code, performs target machine specific optimizations, and finally outputs the object code to be run on the target machine [12].

Compilers perform two types of optimization: machine independent and machine

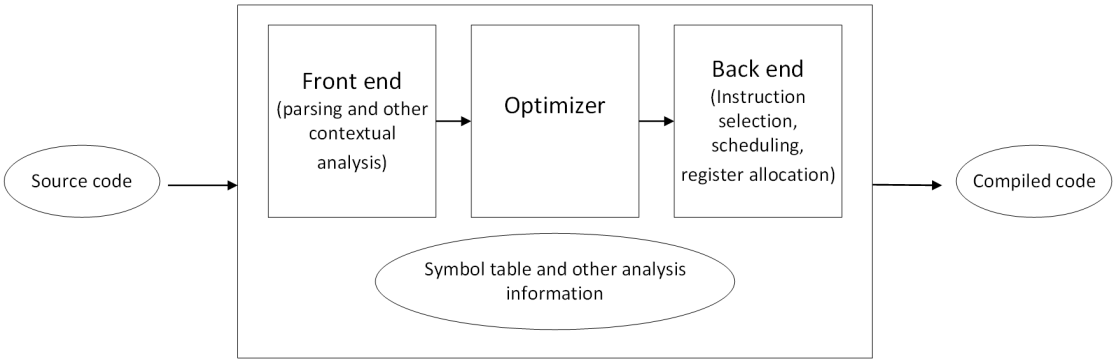

Figure 2.6: General architecture of a compiler [12]

dependent optimization [12]. Machine independent optimization are not dependent on the architecture of the device. The compiler processes the intermediate code and transforms it to a code that do not involve any registers or absolute memory locations [12]. Instead, machine dependent optimization involves the knowledge about the device architecture and tries to get the maximum advantage of the memory hierarchy of the specific target [12]. Machine dependent optimization happens after the code is transformed based on the specific target architecture. Machine dependent optimization involves CPU registers and may also include references to the absolute memory locations [12].

Compilers support different levels of optimization. Table 2.1 represents the optimization levels in TMS320C6000 DSP compiler.

Compilers have proved to be effective optimization method for RISC-type processors. However, the irregular datapaths, small number of registers [18], non-homogeneous register sets, very specialized registers, very specialized functional units, restricted connectivity and limited addressing [19] are a challenge to compilers to produce efficient code. Despite those challenges, DSP compilers perform very well in optimization, and can outperform even the best assembly programmers [12].

One important software development and optimization related unity is the choice of the programming language and the optimization level of the programming language itself. Some of the programming languages used in real-time software development include Ada, C, C++, C, Java and real-time Java [7]. The choice of the programming language itself is very dependent on the company policies and the application itself. As the complexity of embedded and digital-signal processing applications grow, it is effective way to decrease the development costs by utilizing HLLs in programming, and to program only the most time-critical code in assembly. It is showed that HLLs massively lower the development and maintenance costs in embedded systems [18]. From the real-time performance point of view in the DSP application, it is optimization task itself to optimize the usage of the programming

Level	Description
-O0	* Performs control-flow-graph simplification * Allocates variables to registers * Performs loop rotation * Eliminates unused code * Simplifies expressions and statements * Expands calls to functions declared inline
-O1	* Performs all –O0 optimisations * Performs local copy/constant propagation * Removes unused assignments * Eliminates local common expressions
-O2	* Performs all –O1 optimisations * Performs software pipelining * Performs loop optimisations * Eliminates global common sub-expressions * Eliminates global unused assignments * Converts array references in loops to incremented pointer form * Performs loop unrolling
-O3	* Performs all –O2 optimisations * Removes all functions that are never called * Simplifies functions with return values that are never used * Inlines calls to small functions * Reorders function declarations so that the attributes of called functions are known when the caller is optimised * Propagates arguments into function bodies when all calls pass the same value in the same argument position * Identifies file-level variable characteristics

Table 2.1: Example of compiler optimization levels in TMS320C6000 DSP compiler. [16]

language. The three states of the programming language optimization are the effective usage of the language itself, the usage of the DSP-specific programming language extensions, and the usage of the machine level code [20]. All of the levels have their own advantages and disadvantages.

Implementing the algorithm using only the core language C without using any machine level code or language extensions is the fastest to implement and the easiest to reach bit exact results. It also retains the code portability across different platforms, even though it might not be necessary in embedded DSP applications. However, without any DSP-specific language extensions or machine level code, the compiler behaviour might be unexpected, leading to very inefficient performance. It might also waste some DSP resources, leading to non-optimal utilization of the core features. The problem is that HLLs, such C, are not expressive enough for special purpose processors, as they are designed for common architectures. Those regular architectures did not have multiple memories, fixed point computation requirement or modulo addressing modes, so compilers had insufficient information in the source code in order to generate optimal code for target. [21]. That was the reason for the development of the language extension.

If language extensions, like DSP related C-intrinsics are embedded into the software, the performance will be much more optimized. Language extensions can usually be utilized on the core language level without requiring to dive into machine level code, which makes it easier for the developer to utilize them. Also, the compiler is responsible for local frame, register allocation and parallelism when utilising language extensions, which enables higher core functionality utilization, and also makes them a rather simple optimization method for the developer. Even though the compiler makes the optimization, developer has some control over how the compiler will optimize programs [21]. However, language extensions most likely damage the code portability, and they require some knowledge of the instruction set and architecture of the platform. Even though language extensions can be used on the higher language level, it still also slows down the development process.

Machine level code optimization is the only guaranteed way to reach the optimal performance. It makes it possible to fully utilize all the core features. However, the code portability and reusability is certainly damaged, it requires very expert and deep knowledge of instruction set and architecture of the platform, and requires lot of time and perseverance to develop. From the business point of view, it might not be the optimal way of develop DSP software, if adequate performance can be reached using only language extensions. [20]. Of course, some of the key functionalities can be programmed using machine code sequences, and rest of the functionalities can be programmed using language extensions.

Because compilers do not perfectly optimize code generated in HLLs, and because software development is a matter of time and cost, it is becoming com-

mon practice to develop the full algorithm in HLLs and then rewrite the most performance-critical routines in assembly [22]. This results tight in a coupling between the HLL and assembly portions, and makes a mixed programming environment attractive.

As studied in [21], DSP-specific language extensions can significantly improve compiler optimization, and thereby improve the overall application performance, both in the name of the code size and the execution speed.

2.5 CEVA-XC4500 DSP

CEVA-XC4500 is a DSP based on a very-long instruction word (VLIW) model combined with single instruction multiple data (SIMD) concept. This benefits the CEVA-XC4500 with a high level of parallelism and a high code density. CEVA-XC4500 architecture is based on a load and store computer architecture utilizing RISC operations and instructions only. The architecture has dedicated load and store and load units responsible for loading and storing data from and to the data memory directly to and from the registers. All other computation instructions always utilize those registers as sources and destinations. CEVA-XC4500 instruction set is designed with 16 bits, 32 bits, 48 bits and 64 bits wide. [23].

Figure 2.7 represents a block diagram of the CEVA-XC4500 DSP. It consists of general computation unit (GCU), the data address and arithmetic unit (DAAU), the program control unit (PCU), two vector computation units (VCUs), power scaling unit (PSU), memory subsystem and emulation interface. The PCU is responsible for aligning the instructions from the program memory and dispatching them to the different units. It is also responsible for the correct program flow, manages the program counter and various mechanisms for different types of non-continuous instructions. The PCU also supports core emulation and profiling through a standard JTAG-interface. The GCU is responsible for all of the general computations and bit-manipulation operations which are non-vectorized digital-signal processing operations. The DAAU controls all data memory accesses. It has two separate units capable of loading and storing from and to the data memory using different kind of addressing modes. Two VCUs are responsible for all vector computation and vector bit-manipulation operations.

The CEVA-XC4500 has three vector processing units, VA, VB and VM, and two load and store units LS0 and LS1 that are capable of loading and storing vector data. The VA Unit is dedicated mostly to multiply operations and permutations The VB Unit is mainly used for shifts, min and max operations, transposing and bit manipulations. It supports scaling, normalization and packing. It also supports addition and substraction operations. The VM unit is mainly used for post-processing for the output of the VA unit. Load and store units are capable

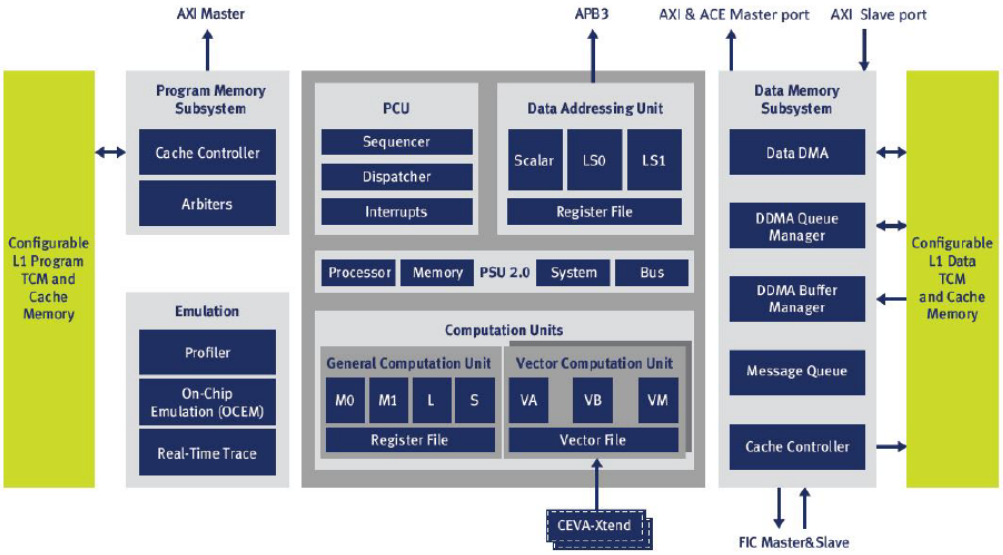

Figure 2.7: CEVA-XC4500 block diagram [23].

of processing 512 bits per cycle, which is 256 bits per VCU. All these five units can work simultaneously. In addition to these vector units, the CEVA-XC4500 has other scalar units to enable running complex code by using the required instruction on each cycle. The vector units are split into two VCUs, each one containing its own vector registers. While the two VCUs, referred as VCU0 and VCU1, are working on different data, the instructions they run are the same. [24].

Figure 2.8 represents the CEVA-XC4500 VCU block diagram. One VCU consists of three different vector processing units VA, VB and VM, and a vector register file called VRF for storing input and output vectors of data. Those vector data registers are called shortly as vectors. One VRF consists of twelve 256-bit input vector registers, four 320-bit vector registers, and four 32-bit vector registers.

The CEVA-XC4500 is a vector core based on SIMD property, which means that a single instruction launches identical operations over a vector of data. In order to efficiently utilize the vector processing capability, inputs and outputs should be organized in a memory. If data for an algorithm is a set of scalars scattered in the memory, it cannot be loaded or stored efficiently with full bandwidth for one cycle. Similarly, if in a loaded vector only a few entries are subject to processing, SIMD operations cannot be used with full power. [24].

The CEVA-XC4500 has a long interlocked pipeline with up to five stages for the VCU execution part, and up to three stages for GCU one. This means that a result of one VCU instruction is available for another VCU instruction with a

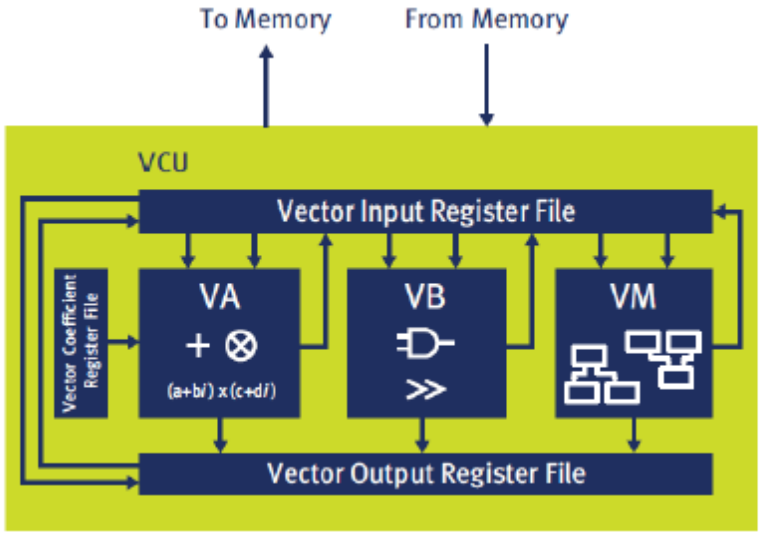

Figure 2.8: CEVA-XC4500 DSP VCU block diagram [23].

delay that constitutes 0 to 4 cycles depending on a sequence used. [24].

The CEVA-XC4500 is a superscalar core with eight-way 256-bit VLIW. It means that up to eight instructions from different units can be executed in parallel for one cycle. Unit instructions can be of 16-, 32-, 48-, 64- or 80-bit width, but an instruction packet composed of them cannot exceed 256-bit length. [24].

The CEVA-XC4500 provides mechanisms of scalar and vector predicates. A scalar predicate is one bit value that defines if an instruction should be disabled to be processed or not. Similarly, a vector predicate is a vector of bits defining which data entries should be processed during the vectorized operation. With scalar predicates, execution of almost all GCU and load or store instructions can be enabled or disabled in runtime. The main purpose of scalar predicates is to avoid branch instructions and to keep sequential code execution. Conditional clauses lead to code branching, realized via conditional jumps and calls. When a conditional branch instruction enters the CEVA-XC4500 processing pipeline, there is four-cycle delay until it reads the condition. By default, the CEVA-XC4500 assumes that the jump will not be taken, so result does not break sequential pipeline operation. However, when this conditional assumption is wrong, then the sequencer is stopped and flashes its pipeline, resulting in penalty of 3 to 7 cycles. A code with many conditional clauses will lead to many occurrences of cycles loss. In order to minimize this loss, the CEVA-compiler checks whether predicted instructions of two alternative branches can be merged, and whether the combined code length will cost fewer cycles than a code separated by branching. [24].

With vector predicates, execution of each atomic operation of a VCU SIMD

instruction can be controlled in runtime. Atomic operation refers to a single operation in a vector of operations of multiple data elements. For example, VCU_add intrinsics explained in the Table 4.5 in section 4 performs 16 atomic operations for 16 data pairs from vectors a and b.

Each VCU instruction can be conditioned on a 16-bit predicate register, whose bits will control every atomic operation of the instruction A zero predicate bit does not prevent execution of the atomic operation itself, but merely blocks writing its result to the destination register. Thus, a word or double-word part of the destination vector is defended by a zero predicate bit and the part preserves its input value. [24]. A word in the context of CEVA-XC4500 refers to a signed or unsigned 16-bit value, and double-word refers to a signed or unsigned 32-bit value. A word part, defined as LOW or HIGH in the intrinsics explanations in the Tables 4.5 and 4.6 in the chapter 4, refers to the most significant (HIGH) or to the less significant (LOW) bits of a double-word.

In the CEVA-XC4500, a generic loop construction requires code and cycles for managing the loop condition and branching. The for–loop is a special case when the maximum number of iterations is known beforehand, and there is an explicit loop counter. The CEVA-XC4500 core's sequencer is equipped with the so called block repeat mechanism, which supports zero-overhead for-loops. In the zero-overhead loops, the loop counter and branching is managed by the hardware, and it does not cost cycles and code. CEVA compiler also supports loop unrolling mechanism. [24].

Chapter 3

Artificial neural networks

This chapter introduces the basics of artificial neural networks. The purpose of this thesis is to produce speed-efficient neural network implementation on a digital-signal processor. Therefore, the comprehensive understanding of neural network architecture is necessary. Section 3.1.1 introduces McCulloch-Pitts neuron, which was one of the first artificial neuron models, trying to mimic the behaviour of a biological neuron. Section 3.1.2 discuss about Rosenblatt's perceptron, which is the most common artificial neuron model today. Multilayer perceptrons (MLPs), including their training and optimization methods, are discussed in section 3.2.

3.1 Introduction

An artificial neural network (ANN) is a massively parallel distributed processor, which can store experimental knowledge and utilize it later [25]. It employs a massive inter-connection units called neurons. Knowledge is obtained by learning from the input data presented to the network. Inter-neuron connection strengths are known as synaptic weights, or shortly as weights. Synaptic weights are used to store the knowledge.

Artificial neural networks are usually defined by the following four parameters [26]:

- Type of neurons
- Neuron connection architecture
- Learning algorithm
- Recall algorithm

Type of neurons defines which kind of intra-network calculation units neural network consists of. These calculation units are called as neurons or nodes [26]. One of the most widely used neuron type is Rosenblatt's perceptron, discussed in section 3.1.2. Rosenblatt's perceptron is commonly referred as a perceptron. Other commonly used neuron types are perceptron's simple predecessor McCulloch-Pitts neuron, introduced in section 3.1.1, and fuzzy neuron discussed in [27]. Common to all of these neurons is that they all originated from the model of a biological neuron, even though neural networks have gradually evolved into purely engineering tools having less and less meaning for real biological neurons [25].

Connection architecture defines the connections between neurons, which is the topology of the ANN. Neurons in the network can be fully connected or partially connected. Fully connected neuron is connected to every neuron in the previous layer, and each connection has it's own weight coefficient. Partially connected neuron is connected to only a few neurons in the previous layer. Fully connected networks are typically utilized with MLPs, discussed in section 3.2, while partially connected neurons are typical in convolutional neural networks. Convolutional neural networks are not discussed in this thesis, as they are typically applied in image recognition applications. In addition to the connection type of the neurons, the connection architecture can be distinguished depending on the number of input and output neurons, and depending on the types of layers used. Connection architecture can be [26]:

- Autoassociative or heteroassociative architecture
- Feedforward or feedback architecture

In an autoassociative network architecture, input neuron of the network are also output neurons. Autoassociative refers to a memory system, which is capable of restoring full piece of data, even when only a tiny portion of that piece of data is reprented. One example of autoassociative network is Hopfield network, popularized by J. Hopfield in [28], as it is capable of remembering data by observing only a portion of it. In a heteroassociative network architecture, there are separate input and output neurons. MLPs and Kohonen networks [29] are type of networks with heteroassociative neurons.

Furthermore, depending on the existence of feedback from the neuron output back to the input, architecture can be determined to be feedforward or feedback architecture. In feedforward architecture, there are no feedback connections involved, and neurons do not remember their previous output values or states. MLPs are type of feedforward networks. In a feedback architecture, there exists connections back from the output to the input, and such network holds in memory its previous states [26]. The next state depends on the current input and the previous states

of the network. Hopfield network [28] is type of network with feedback loops. Feedback neural networks are often called recurrent neural networks (RNNs) [26].

Learning algorithm is an algorithm which trains the network. The fundamental feature of neural networks is their ability to learn knowledge from input data, and this would not be possible without learning algorithms. Learning algorithms are typically classified into supervised learning, unsupervised learning and reinforcement learning [26]. Learning algorithms are further discussed in section 3.3.

Recall algorithm is an algorithm, which extracts learned knowledge from the neural network [26]. When new data is fed to the trained neural network, the recall algorithm provides the corresponding output based on the learned knowledge. Recall of the neural network is often called as a neural network inference.

3.1.1 McCulloch-Pitts neuron

Figure 3.1 represents a model of McCulloch-Pitts neuron (MPN), introduced in [30]. This model of a neuron tries to mimic a real biological neuron, even though it is highly simplified version. In [30] it is shown how to encode any logical proposition by an appropriate network of MPNs. Therefore, theoretically anything that can be done with a computer can also be done with a network of MPNs. It is also shown in [30] that every network of MPNs encodes some logical proposition. So if the brain were a neural network, then it would encode some complicated computer program. But the MPN is not a full model of a biological neuron, it is only a highly simplified model of it. Still, MPN is very important basis for all of the modern artificial neuron models.

In Figure 3.1, a model on MPN is represented. x_1-x_3 are boolean valued inputs to the neuron, representing transmission channels (dendrites) of a biological neuron. Like in biological neurons, the input values can be inhibitory or excitatory. Small white circle at the end of input x_3 in Figure 3.1 represents the inhibitory input, meaning that the input value is complemented, while x_1 and x_2 are exhibitory inputs. g and f represent a processing unit of the neuron, representing the soma in biological neurons. Function g aggregates the input signals to a single numerical value, and function f produces the output by taking the output of g as an input. The function f will output the boolean value 1 if the aggregation performed by the function g is greater than some threshold value b, otherwise it will return 0. y represents the boolean output of the neuron, acting like an axom in biological neuron. Mathematically MPN can be described as follows:

$$y = f(g(x)) = f(\sum_{i=1}^{n} x_i) = \begin{cases} 0, & \text{if } g(x) < b \\ 1, & \text{if } g(x) \geq b \end{cases} \tag{3.1}$$

where $g(x)$ is the aggregation of inputs x_1-x_n, $f(x)$ is the function which produces

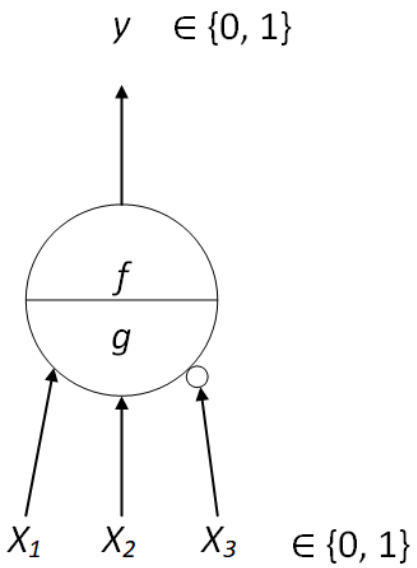

Figure 3.1: Representation of McCulloch-Pitts neuron model.

output y based on $g(x)$ and threshold b. $f(x)$ acts like activation function in perceptron model (Figure 3.2), and it is type of hard limiter (equation (3.6)).

The MPN model is simple, but it has substantial computing potential and also a precise mathematical definition. However, it only generates a binary output and also the weight and threshold values are fixed. MPN need to be set up correctly for model to correspond to the desired definition of the categories. MPN does not involve any kind of learning, even though it is based on the biological neuron. Thus, the neural model with more flexible computational features needed to be obtained, and that is the reason for the invention of Rosenblatt's perceptron model, discussed in section 3.1.2.

3.1.2 Rosenblatt's perceptron

Rosenblatt's perceptron, introduced in [31], is the basic building block for all modern neural networks [32]. In this thesis, the Rosenblatt's perceptron is called shortly as perceptron, or an artificial neuron. The perceptron is based on the idea of neuron model called McCulloch-Pitts Neuron (MPN), first introduced in [30] by McCulloch and Pitts. It is based on the model of biological neuron which receives neural signals from other neurons, and produces a response. A perceptron is essentially an algorithm for supervised learning of binary classifiers. Mathematically, it takes a weighted aggregate of its inputs, applies a function and produces an output [31]. Supervised learning means a task of learning input-output

mapping from example input-output pairs. A binary classifier is a function which can decide if an input vector belongs to a some specific class or not. A fully trained perceptron can perfectly classify input patterns if they are linearly separable [25]. A single perceptron is the most simplest version of ANNs, as it consist only one neuron on a single layer, the perceptron itself.

Figure 3.2 represents the mathematical signal-flow graph of the perceptron. The perceptron consists of three basic elements [25]:

1. A set of connecting links, each of which is characterized by a weight.
2. An adder for summing the input signals, weighted by the respective synaptic weights of the neuron.
3. An activation function for limiting the amplitude of the output of the neuron.

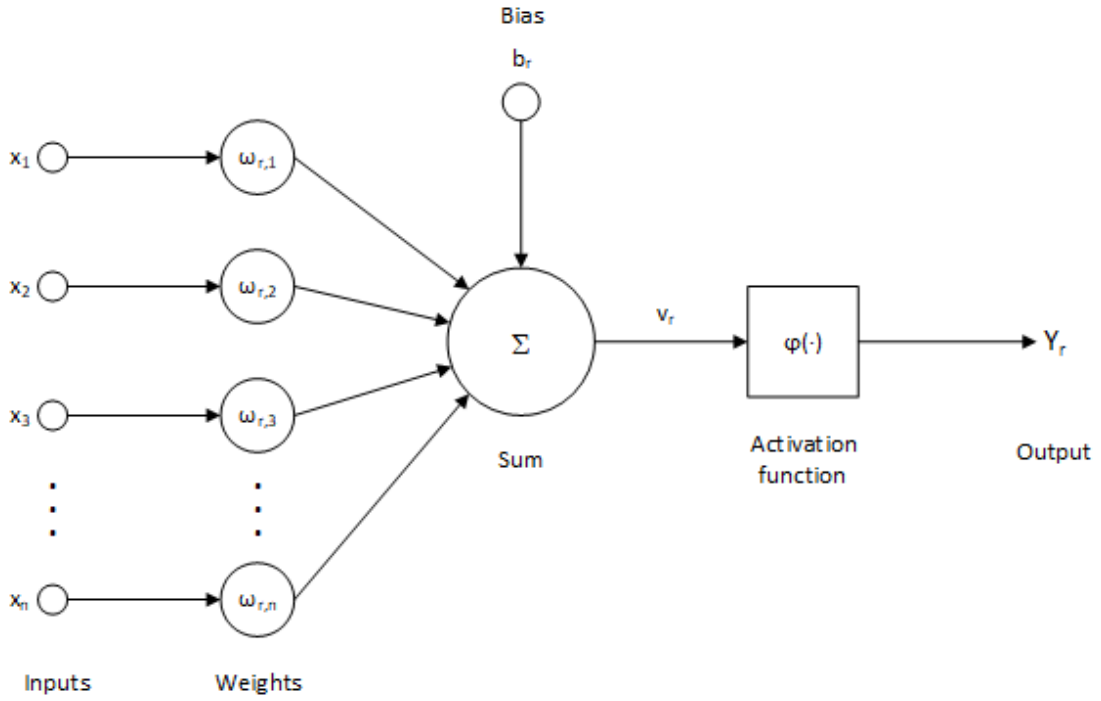

Figure 3.2: A nonlinear model of an artificial neuron called perceptron, labeled r.

In more details, the perceptron consists of n real valued inputs, labeled as x_1 - x_n. Each of the inputs have corresponding synaptic weight, labeled as $w_{r,1}$ - $w_{r,n}$ respectively. These input-weight pairs are aggregated together in order to form u_r. As the Equation (3.2) represents, the aggregation is a weighted sum on input-weight pairs, and it represents the potential which is induced by other neurons or inputs.

$$u_r = \sum_{i=1}^{n} \omega_{r,i} x_i \tag{3.2}$$

In the perceptron model, there is also one special input called bias b [26]. Bias has a constant weight value of 1. It can be also considered in a way that the perceptron has one constant input of 1, which is weighted by the bias coefficient b. Mathematically speaking, the purpose of the bias is to adjust the activation threshold of the perceptron, thereby increasing the flexibility of the perceptron model. Bias represents the background potential of the neuron. u_r and bias b together form the induced local field of the neuron v_r, represented in Equation (3.3).

$$v_r = u_r + b_r \tag{3.3}$$

As mentioned earlier, this one special input value can also be considered as constant input value of 1, paired with the weight b. In many cases this constant input can be considered to be input x_0, and the bias b can be similarly assigned to be $w_{r,0}$. In this case, b_r can be embedded to the aggregation, as Equation (3.4) illustrates.

$$v_r = \sum_{i=1}^{n} \omega_{r,i} x_i + b_r = \sum_{i=1}^{n} \omega_{r,i} x_i + \omega_{r,0} x_0 = \sum_{i=0}^{n} \omega_{r,i} x_i \tag{3.4}$$

Activation

In order to produce the final output y_r of the perceptron, the induced local field v_r is activated with a function called activation function [32] or squash function, $\varphi(.)$:

$$y_r = \varphi(v_r) \tag{3.5}$$

The purpose of the activation is to limit the amplitude of the neuron output, and to add non-linearity to the inference [32]. If linear activation function, like identity activation $\varphi(v) = v$ is used, multiple perceptron layers (discussed in section 3.2) are equivalent to single perceptron layer. In the case of linear activation, extra layers do not add free parameters to the network. In addition to non-linearity, the activation function is desirably continuously differentiable. Without continuous differentiablity, gradient-based learning methods (discussed in section 3.3) are not possible, or they might have problems to progress with learning. Few of the most commonly used activation functions are:

- Threshold function or hard limiter

$$\varphi(v) = \begin{cases} +1 & \text{if } v \geq 0 \\ 0 & \text{if } v < 0 \end{cases} \tag{3.6}$$

Threshold function outputs only two values, either +1 or 0. It is symmetric function, and its breakpoint can be easily moved with bias. However, it is neither continuous or continuously differentiable. It was used in many early neural network models, and is fundamental part of the McCulloch-Pitts neuron model [30], discussed in section 3.1.1.

- Rectified linear unit (ReLU)

$$\varphi(v) = \max\{0,\, v\} \tag{3.7}$$

The rectified linear unit outputs the input itself, if the input value is positive, and zero otherwise. Therefore, it is linear for half of the input domain, and non-linear for the other half. Linear property make it easy to optimize with gradient based methods [33]. The gradient is 1 for all positive values, and 0 for negative values. This means that during backpropagation training, discussed in section 3.3.2, negative gradient will not be used to update the weights. Because the gradient is 1 for all positive values, the training speed is very good compared to other non-linear functions due to good gradient flow. ReLU also preserves many of the properties that make linear model generalize well [33]. It has become the default activation function for many types of neural networks, as it is easy to optimize, it typically convergences fast, and it has high computation speed [34].

- Sigmoid function

$$\varphi(v) = \frac{1}{1 + e^{-av}},\ a \in \mathbb{R} \tag{3.8}$$

The sigmoid function is commonly used S-shaped nonlinear activation function. It is commonly used on the output layer neurons, as it produces restricted output between 0 and 1. This outcome can be interpreted as probability of that output. Using design parameter a, the shape of the sigmoid curve can be tuned. One advantage of the sigmoid function is that its gradient is well-defined, which is probably one of the reasons for its popularity. Sigmoid functions can be used also on the hidden layer neurons, however, the gradient of the function is very close to zero over a large portion of its domain, which makes it harder and slower for the learning algorithm to learn. This unfavourable feature is often called as gradient vanishing problem [35]. Vanishing gradient make it difficult to know which direction the parameter should move in order

to imporve the model performance [33]. Other commonly identified problem with the sigmoid function is the saturation. The saturation means that large input values snap to sigmoids maximum output value of 1, and small values similarly snap to the 0. Furthermore, the function is very sensitive only around the mid-point of the input, around 0.5 [33]. The limited sensitivity and saturation problem happen whether the input contains useful information or not, and once saturated, it becomes very challenging for the learning algorithm to continue improving the model performance.

- Softmax

$$\varphi(v)_i = \frac{e^{v_i}}{\sum_j e^{v_j}} \tag{3.9}$$

Softmax is utilized in the output layer of the neural network to detect multiclass classification probabilities. It squashes output vector in the range (0, 1), and all the resulting elements add up to 1. The Softmax function cannot be applied independently to each element, as it's value depends on all elements of the output vector. If the classification probability is not important in the inference, the softmax activation can be replaced with argmax activation.

Learning

Learning is fundamental property of perceptron and ANNs. Perceptron can acquire knowledge from the training set, and store it in its weight coefficients. Via learning, perceptrons are suitable both for classification and regression problems.

The perceptron rule is a sequential learning procedure for updating the perceptron weights. In other words, it is the way of learning perceptron input-output mapping for a classification problem. The learning rule is an example of supervised training, in which the learning rule is provided with a set of examples of proper network behavior called a training set $S = \{X(0), d(0)\}, \quad \{X(1), d(1)\}, \quad ... \quad \{X(n), d(n)\}$, where $X(n)$ is the example input vector and $d(n)$ is the corresponding output value.

The Perceptron convergence theorem [36] states that for any data set which is linearly separable, the perceptron learning rule is guaranteed to find a solution in a finite number of steps. In other words, the perceptron learning rule is guaranteed to converge to a weight vector that correctly classifies the examples provided. However, it is important to understand that the converged weight vector W' is not necessarily unique, and it depends on the initial weight vector W. Also, the converged weight vector W' is not necessarily same as the optimal solution $W*$, even though it correctly classifies the learning examples. A function is said to be linearly separable when its outputs can be discriminated by a function which is a linear combination of features. The convergence theorem also states that the

Algorithm 1 Perceptron learning rule

```
Choose a learning rate η > 0
Choose a random weight vector W
Training set S(n)
i = 0
while i < n do
    if d(n) ≠ y(n) then
        e(i) = d(i) − y(i)
        W = W + ηe(i)X(i)
        i = 0
    else
        increment i
W' = W
```

learning rate η can be chosen to any positive value, and the perceptron learning rule will find a solution. However, choosing the learning rate properly will dictate how fast the learning rule will converge to a solution.

3.2 Multilayer perceptrons

Multilayer perceptrons (MLPs), also called as feedforward neural networks or deep feedforward networks [33], are neural networks with one or more hidden layers. As the name suggest, they consist of multiple perceptrons, discussed in section 3.1.2. A hidden layer is a container of perceptrons between the input and output layers. A single perceptron has a fundamental limitation of being able to classify only linearly separable patterns, while MLPs can approximate any continuous function [25]. The goal of the MLPs is to approximate some function $f*$. A multilayer perceptron is considered feedforward, because the information flows through the network from the input, through the immediate computations in hidden layers, finally to the output, without any feedback connections. When feedforward neural network is extended to contain feedback loops, they are called recurrent neural networks (RNNs). [33]

Multilayer perceptron features can be highlighted with three basic elements [32]:

1. The model of each perceptron in the network includes a nonlinear activation function that is differentiable

2. The network contains at least one layer that is hidden from both input and output nodes

3. The network exhibits a high degree of connectivity, the extent of which is determines by the perceptron weights of the network

Figure 3.3 represents the model of a multilayer perceptron. It consists of a input layer, hidden layers and an output layer. The input consists of an input vector $X = [x_1, x_2, ..., x_n]^T$ of $n \in \mathbb{N}^+$ values , which is forwarded to the first hidden layer. There are $m \in \mathbb{N}^+$ hidden layers, h_1-h_m, each having individual number of perceptrons. The layer h_m has k perceptrons, named $h_{m,1}$-$h_{m,k}$. Lastly, there is an output layer, consisting $j \in \mathbb{N}^+$ perceptrons $h_{y,1}$-$h_{y,j}$. The output layer produces the output vector $Y = [y_1, y_2, ...y_j]^T$ of the MLP. It depends on the application how the output vector information is utilized. Each of the individual perceptrons in the network are modelled as described in section 3.1.2.

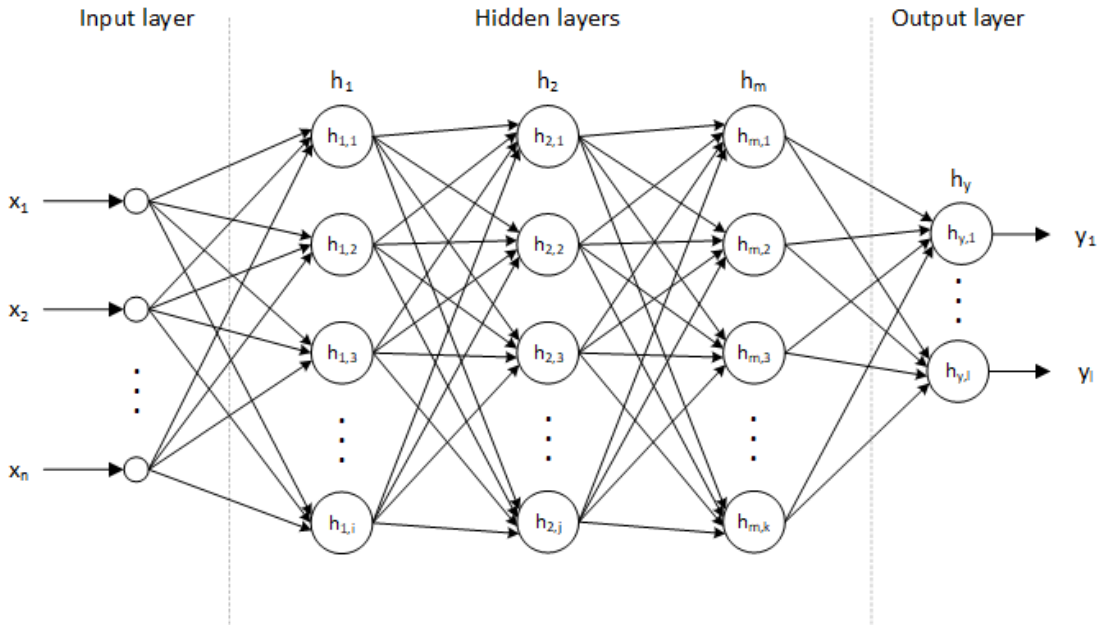

Figure 3.3: A model of an artificial neural network called multilayer perceptron.

3.3 Training procedure

The usefulness of artificial neural networks is based on the idea of acquiring knowledge through a learning process. Intra-neuron connection strengths, known as synaptic weights, are used to store the acquired knowledge. Learning algorithm is the algorithm which trains the neural network by adjusting the weight values. Learning algorithms are classified into three groups:

1. Supervised learning
2. Unsupervised learning
3. Reinforcement learning

Supervised learning is the machine learning task of learning a function that maps an input vector to an output value based on example input-output pairs. In supervised learning, the training data samples consists of input vector X, and the desired output vector Y. The training is performed until the network learns to associate each input X to its corresponding output Y. It approximates function f^*, so that $Y = f^*(X)$. Figure 3.4 represents the supervised learning methodology. Supervised learning happens in three steps:

1. The neural network is fed with input data. The network, named as learning system in the Figure 3.4, produces corresponding output. The input is fundamentally one vector sample describing the state of the system to be approximated.
2. The teacher will tell to the network, based on the input data, what the expected output should be. Error signal is generated based on the difference between this expected output and the neural network output.
3. The weights are adjusted by the generated error signal.

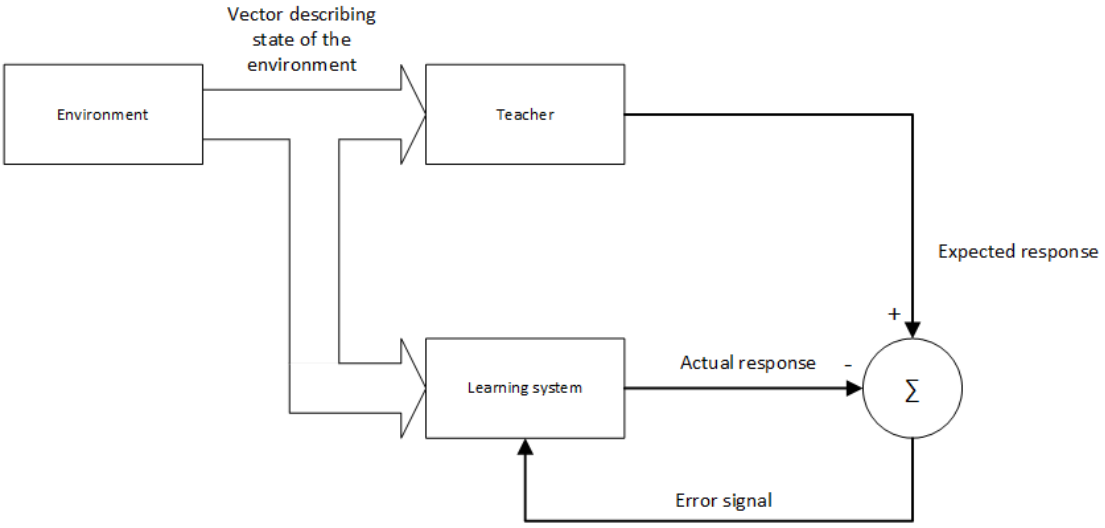

Figure 3.4: Block diagram of learning with a teacher. [32].

Unsupervised learning is a training method, in which only training input X is supplied to the network. The network learns some internal feature of the whole set of all data inputs, but there are not any desired output. Unsupervised algorithms are usually further divided into competitive and non-competitive algorithms.

Reinforcement learning is a method, in which the input vector X is presented to the network, and it is allowed to determine corresponding output. If the output is good in the names of cost function C, the existing neural weights are increased (rewarded), otherwise they are decreased (punished).

3.3.1 Cost function

Cost function C is a function that measures the performanse of a machine learning model for given data. Cost function quantifies the error between predicted output and expected output values. Cost function presents this error in the form of a single real valued number. The purpose of the cost function is to be either minimized or maximized. If the cost function is to be minimized, the returned value of the function is usually called cost, lost or error. The goal of the training is to find a model which minimizes the value returned by the cost function. If the purpose of the cost function is be maximized, the return value is typically called a reward. In this case, the purpose of the training is to find a model for which the returned cost function value is as large as possible. [25].

Two very commonly used cost functions are mean absolute error (MAE) and mean squared error (MSE). Equation (3.10) represents the cost value C in MAE function, and equation (3.11) represents the cost value in MSE. C states for the return value of a cost function, n is the total number of data samples, $d(i)$ is the expected output for the ith data sample, and $y(i)$ is the model output for the ith data sample.

$$C = \frac{1}{n} \sum_{i=1}^{n} |d(i) - y(i)| \tag{3.10}$$

$$C = \frac{1}{2n} \sum_{i=1}^{n} (d(i) - y(i))^2 \tag{3.11}$$

Each cost function treats the difference between observations and expected outcome in a unique way. The distance between ideal result and predictions are having attached a penalty by metric, based on the magnitude and direction in the coordinate system. For example, MAE does not add any additional weight to the error distance, so the error growth is linear. In MSE instead, the error grows exponentially with larger error distances. It adds a massive penalty for prediction that are war away from expectation, and minimal penalty for close predictions.

Binary cross-entropy (BCE) is a loss function used on problems involving binary decisions [37], such as one hot encoding problems. Equation 3.12 represents the loss in BCE. In multilabel problems, where an example can belong to multiple classes at the same time, the model tries to decide for each class whether the example belongs to that class or not. BCE measures how far away from the true binary value the prediction is for each of the classes, and then averages these classwise errors to obtain the final loss.

$$C_{bce} = -\frac{1}{N}\sum_{i=1}^{N}[y_i log(p(y_i)) + (1 - y_i) log(1 - p(y_i))] \tag{3.12}$$

3.3.2 Backpropagation of errors

Backpropagation of errors, or simply backpropagation algorithm, is the most popular learning algorithm rule for performing supervised learning tasks for MLP. Backpropagation algorithm uses a gradient-search technique to minimize a cost function between the desired output and the actual MLP output. The gradient signifies how the error of the network changes with the changes to the network's weights. Backpropagation makes it possible to extend MLP to many layers. The modern version of the backpropagation, also called the reverse mode of automatic differentiation, was first published in [38] without NN context. In [39] the backpropagation algorithm is attached to several different NNs, making them to learn much faster than previous learning approaches, solving problems which had previously been insolvable.

Algorithm 2 describes the backpropagation on pseudocode level. The basic principle is that backpropagation algorithm propagates backwards the error between NN output and desired output through the network. After providing an input pattern, the forward propagated output of the network, the inference, is then compared with a given target pattern, and the error of each output unit is calculated. This error signal $e(n)$, difference between desired output $d(n)$ pattern and inferred output $y(n)$, is propagated backward, and a closed-loop control system is thus established. At the heart of backpropagation is an expression for the partial derivative $\frac{\partial C}{\partial w}$ of the cost function C with respect to any weight w in the network. The partial derivative express how quickly the value of the cost function changes when weights are changed.

The backpropagation requires two assumptions from the cost function. The first is that the cost function can be written as an average $C = \frac{1}{n}\sum_x C_x$ over individual cost functions C_x for individual training examples x. The second assumption is that the cost function is a function of the outputs from the NN, $C = f(y)$

The adjustments of the synaptic weights depends upon the information of the

Algorithm 2 Backpropagation algorithm

Choose a random weight matrix $w(0)$
while Stopping criterion is not met **do**

    %Forward propagation
    for each layer $h_i, i = 1...m$ **do**
        for each perceptron $h_{m,j}$, $j = 1...k$ **do**
            $y_{h_{m,j}} = \varphi(w^T_{h_{m,j}} x_{h_{m,j}})$

    %Backpropagation
    for each layer $h_i, i = 1...m$ **do**
        for each perceptron $h_{m,j}$, $j = 1...k$ **do**
            $\frac{\partial C(w)}{\partial w_{i,i+1}} = \frac{1}{N} \sum_{d=1}^{N} \frac{\partial C(w)_d}{\partial w_{i,i+1}}$
            $\Delta w^{h_i}_{i,i+1} = -\eta \frac{\partial C(w)}{\partial w_{i,i+1}}$

input neuron and the output neuron, and nothing else. The output of the neuron depends upon all the connected neurons. Therefore, the adjustment of the synaptic weight is proportional to both the input signal and the output error.

One hot encoding

In a case of having multiple output neurons for multiple output options, one hot encoding is a method for converting so called category variables into a binary-style of categorizing [40]. Label encoding refers to a method, in which every output category has unique numerical label representing the category. However, this type of categorizing gives different categories natural ordered relationship, giving higher labels higher weight. Therefore, label type of encoding adds unwanted correlation. One hot encoding gives every outcome unique binary representation, which removes the natural ordering property. Output of n neurons can be represented with n-bit encoding, so that the bit that is representing the desired output is set to 1, and other bits are set to 0.

3.3.3 Optimization algorithms

Optimization algorithms are used to minimize or maximise the cost function C of a ML model. The internal parameters are key for efficient and accurate ML models, so various optimization algorithms and strategies are used to train models. Optimization algorithms can be categorised in two groups: first order optimization algorithms and second order optimization algorithms.

The goal with first order optimization algorithms is to minimize or maximize cost function C based on the gradient of the C respect to a certain model variable θ [41]. In the case of NNs, the model variable θ is the weight vector W of the NN. Gradient of the cost function tells whether the function is increasing or decreasing at a particular value of θ.

Similarly, the goal of the second order optimization algorithms is to minimize or maximize the cost function C based on the second gradient of the C respect to the certain model variable θ. The second order derivative of the C tells wheter the gradient of the C is increasing or decreasing at a particular value of θ. The second order derivative is computational heavy to calculate, therefore, the second order optimizations are not as widely used methods as the first order optimization algorithms.

Gradient descent is a first order optimization algorithm commonly used in NN models. Gradient descent minimizes the cost function by moving iteratively to the direction of the steepest descent as defined by the negative of the gradient. Equation (3.13) represents the weight update rule of the MLP with gradient descent. η is called as learning rate, which defines how big steps are taken when updating the weights based on the gradient of the cost function.

$$w(n+1) = w(n) - \eta \nabla C(w(n)) \tag{3.13}$$

An advanced form of the gradient descent, stochastic gradient descent (SGD), is the most used optimization algorithms for ML in general [33]. SGD have also lots of variants, for example Adam optimizer, adaptive gradient algorithm, and root mean square propagation [42]. In the gradient descent algorithm, all the samples in the training set are run in order to do a single update for a parameter in a particular iteration, whereas in SGD, only one sample or subset of training sample of the training are used to do the update for a parameter in a particular iteration. If a subset of samples are used, the method is called minibatch stochastic gradient descent [33]. As gradient descent is based on the derivative of the cost function, in order to optimize cost function with it, also activation functions in the NN must be differentiable.

Adam optimizer is one extension of SGD based optimization algorithms. It combines properties of the adaptive gradient algorithm (AGA) and root mean square propagation (RMSP) to provide an optimization algorithm that can handle sparse gradients on noisy problems.[43] It computes adaptive learning rates for each parameter. Adam uses both the average of the first moment and average of the the second moment of the gradient when adapting learning rates. The Adam calculates an exponential moving average of the gradient and the squared gradient, and two tuning parameters control the decay rates of these moving averages. [42].

3.3.4 Regularization

Regularization is a technique to penalize the size of the weights of a NN. Regularization penalizes strong weight values, so that strong output confidence cannot be achieved from the output of just single single neuron. The strong output confidence can be achieved only from the consensus of multiple neurons. Since many neurons have to agree to achieve a strong output value, the output value is less likely to biased just by a single neuron. It especially helps to prevent overfitting, as adjusting the model to be more complex requires bigger weight values, and that is penalized.

There are two commonly used regularization methods, L1 and L2 regularization. A regression model that uses L1 regularization is called Lasso Regression [44], and model which uses L2 regression is called Ridge Regression [45].

The L1 regularization penalizes the absolute size of the weight. Equation (3.14) represents the L1 regularization element. Here λ is the tuning parameter that decides how much the flexibility of the model is penalized. w represents a weight coefficient of the model. In L1 regularization, most of the weights will be zero, and only some medium or large size weights remain [44]. Because L1 regularization tends to produce sparse weights values, it naturally does a feature selection. Insignificant input features are assigned with zero weights, and significant features are assigned with large weight value.

$$R = \lambda \sum_i |w_i| \tag{3.14}$$

The L2 regularization penalizes the squared size of the weight. Equation 3.15 represents the L2 regularization element. Here λ is the tuning parameter that decides how much the flexibility of the model is penalized. w represents a weight coefficient of the model. L2 regularization encourages very small, but non-zero weights. The inference is established by almost all weights, thus reducing the model bias. There are no neurons that can turn around model output by themselves. As L2 regularization does not tend to make weights zero, it is not robust to the outlier data samples.

$$R = \lambda \sum_i w_i^2 \tag{3.15}$$

Chapter 4

Implementation

In this thesis, fully-connected feedforward neural network inference is implemented on the CEVA-XC4500 DSP. The feedforward neural network architecture is potential ML algorithm for an DSP application due to the parallelization potential, and due to promising results in various classification problems. For the design of advanced neural network based real-time ML application, it is essential to understand the theory behind neural networks, how to collect suitable data for the model training and validation, how to create a working model, and how DSP architecture can be efficiently utilized. As discussed earlier in this thesis, the first and the most important optimization for an DSP algorithm is to design the algorithm itself from the DSP architecture point of view. For that reason, it is important to keep the DSP in mind when creating the neural network model itself.

Theory behind feedforward neural networks, supervised learning, and DSP algorithm development and optimization methods are explained in previous chapters, while this chapter focuses on the neural network model creation and the DSP implementation for a specific use case on a specific DSP. At first, the use case is shortly explained. Next, NN model creation is discussed including the data collection, the use case specific data-analysis, hyperparameter tuning, and the model validation. Later, DSP related implementation and optimization methods are discussed for the specific use case. Evaluation of the implementation is represented in the chapter 5.

Because the source codes of the NN model and the DSP implementation are confidential property of Nokia, only pseudocodes are represented in this thesis.

4.1 Use case

The use case is to implement NN-based real-time symbol decoder for the 5G uplink control information. The implementation is executed on the Nokia 5G baseband

product, utilizing CEVA-XC4500 DSP.

General wireless communication system is represented in the figure 4.1. The general wireless communication system consists of a transmitter, a channel and a receiver. In a general system, the transmitter and receiver are designed and optimized to mitigate the detrimental effect of multipath fading channels. Different channel effects are modelled as stochastic processes whose probability distributions tend to mimic the behaviour of a real waveform travelling through the medium. This enables to design effective mathematical models close to optimal in many realistic scenarios. However, real systems include imperfections that are not gaussian, stationary or linear. The model design itself might include imperfections, hardware components are imperfect and computationally less complex than ideal. Those non-idealities and imperfections need to be taken into account, causing non-optimal behaviours. Therefore, stochastic mathematical models are not perfect for the modelling of wireless communication scenarios.

Figure 4.1: A simple wireless communication system consisting of a transmitter and a receiver through a channel.

In order to improve the model, ML-based approach is a potential option. ML based approach does not require any mathematical model parameters or any knowledge about the system itself. The design of the model is learned only using the direct measurements from the system data. It is proved in the universal approximation theorem, first introduced in [46], later expanded in [47], that any continuous function on a closed interval can be approximated with infinitely small error using ML-based approach, meaning that any computer algorithm can be approximated using ML. There are lots of ML based software tools and libraries designed to utilize ML approaches, making it easier to find suitable solutions for different applications.

NNs are proved to find, or to be part of the solutions in very complex mathematical problems such as [48] or in the applications described in [49]. They are studied and utilized also in the real-time systems in telecommunication [50]. As explained in the chapter 3, the inference of a neural network is mostly based on the calculation of SOPs, which is also one of the most optimized operations in the DSPs. The traditional stochastic mathematical model based approach of solving given real-time symbol decoding task is consuming relatively high amount of DSP

cycles. Therefore, it is an potential application to try NN-based approach.

Figure 4.2 represents a flowchart of the use case. The upper graph represents an approach with the conventional decoder and the lower graph represents an approach with the NN based decoder. The goal of the given real-time symbol decoder is to map given input pattern consisting of 15 complex valued inputs to one of the four possible outcomes, named with labels 1-4. Those 15 inputs of the decoder are formed from two underlying binary input parameters, represented as input parameter 1 and input parameter 2 in the figure 4.2. Those two underlying input parameters are transformed into 15 input symbols by first encoding, repeating and modulating the parameters. Because there are two underlying binary inputs, the output of the decoder is one out of the four possible combinations of those two inputs.

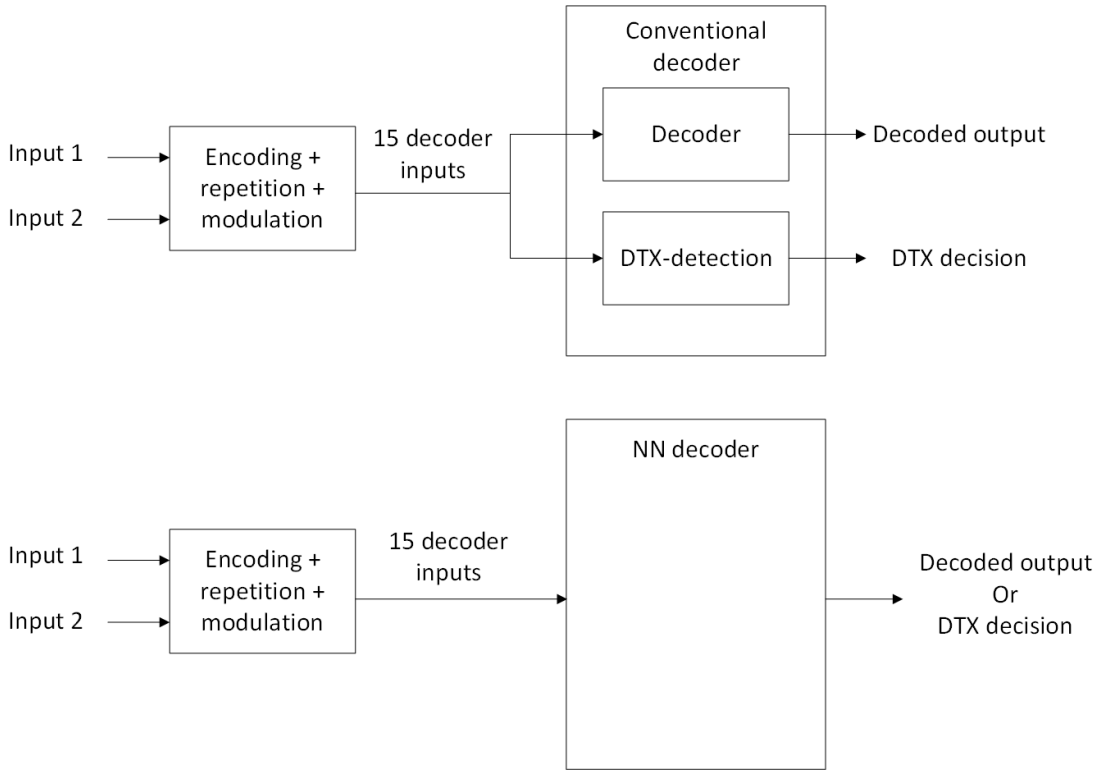

Figure 4.2: Flowchart of the use case. Upper graph describes the conventional decoder consisting of the decoder and DTX-decision separately. The output of the conventional decoder is decoded output and DTX-decision. The lower graph describes the NN based decoder, outputting decoded output or DTX-decision.

In addition to the decoding, the purpose of the decoder is also to detect so called discontinuous trasmission (DTX) case. DTX refers to a situation in which

the receiver detected a signal, but nothing was generated by the transmitter. The input of the decoder is just noise, and it has no meaning. The decoder first checks if the input data is valid non-DTX situation, and after that it decodes the input content. This check is called as DTX-decision and it is represented also in the figure 4.2.

Both decoder approaches use 15 complex valued symbols as an input. As the figure 4.2 illustrates, one main difference between the conventional and NN based approach is that the conventional decoder consists of separate decoder and DTX-detector, whereas NN based decoder does the decoding and DTX-decision simultaneously in the same network. If not separately mentioned, in this thesis the term conventional decoder refers to the combination of the decoder and DTX-detector blocks. In the conventional decoder, the input of 15 symbols is fed both to the decoder and DTX-detector blocks. If the DTX-decision is negative, the decoder output is utilized for further processing. Input-output-mapping of the conventional decoder is represented in the Table 4.1. In the NN based decoder, the output is one out of five classes, where class five represents positive DTX-decision, and classes 1-4 represents negative DTX-decision and corresponding decoded output label. Table 4.2 represents the input-output-mapping of the NN based decoder.

Table 4.1: Input-output-mapping of the conventional decoder. Output of the decoder and DTX-decision are separated.

Conventional decoder			
Input parameter 1	Input parameter 2	Decoder output	DTX-decision
0	0	1	0
0	1	2	0
1	0	3	0
1	1	4	0
Noise	Noise	Random	1

4.2 Neural network model

Data for the NN model is collected using Matlab simulation model. The simulation model is not created for this thesis itself, as it is made for the Nokia mobile networks software development purposes in general. The model is used to fork input values for the NN, the 15 complex valued input symbols represented in the figure 4.2, with the corresponding output labels. Those input-output pairs are used for the supervised learning-method for the training of the NN. The NN model covers and

Table 4.2: Input-output-mapping of the NN decoder. Output and DTX-decision are merged.

NN decoder		
Input parameter 1	Input parameter 2	Decoded output
0	0	1
0	1	2
1	0	3
1	1	4
Noise	Noise	5

combines both the decoder and the DTX-detector blocks from the figure 4.2 in a way the input is the same, but the output is one out of five different options. Four of the outputs represent the decoder output, and one of the outputs refers to the DTX-decision output. One hot encoding, explained in the chapter 3.3.2, is used to train the NN with these five different outcomes. Table 4.3 represents the one hot encoding for the given model and data.

Table 4.3: One hot encoding of the outputs.

Underlying input	Label encoding	One hot encoding
Input combination 1	1	00001
Input combination 2	2	00010
Input combination 3	3	00100
Input combination 4	4	01000
DTX	5	10000

The input data is normalized in order to fit it into a accurate range in fixed-point format. In general, input data normalization is a good practise in order to achieve good results with the classification performance [51]. However, the real-time execution speed is the most important evaluation criterion in this implementation, so it is better not to preprocess the input too much in order not to consume DSP cycles in vain. If the classification performance is on acceptable level without further preprocessing, there is no need to do it. An acceptable level in this case means the same classification performance than in the conventional mathematical model based decoder, which performance is based on the 3GPP specification [52]. Performance of the decoder is measured in false-alarm rate (FAR), bit error rate

(BER) and misdetection propability (MDP). Those metrics are explained in section 5.1.1.

A feedforward neural network is chosen for the machine learning architecture due to its simple calculation structure. NN inference is based on the calculation of SOPs, which are one of the most optimized operations in the DSPs in general. Intuitively, it is good to keep the number of parameters as small as possible, as smaller number of parameters requires less memory, less operations to fetch the data from the memory, and requires less operations to process the data. As discussed earlier in this thesis, it is also important to build the model for the specific DSP in mind, so that the VCUs can be fully utilized. CEVA-XC4500 DSP can hold 512 bits in total in its vector registers, meaning 16 values of 32 bits or 32 values of 16 bits. Due to the SIMD nature of the CEVA-XC4500 DSP, the processing of an operation with 16 data values of 32 bits and only one data value of 32 bits consumes the same amount of cycles. Therefore, it is good to keep NN layer sizes divisible by 16 in order to fully utilize the vector processing capability.

4.2.1 Data

The input of the decoder consists of 15 signed complex valued input symbols of 32 bits. In CEVA-XC4500, signed complex values of 32 bits are loaded into memory as two signed 16-bit values, in a way the real and imaginary parts are separated. In the Matlab-simulation model, the inputs are handled as floating point values, but in the DSP they are handled in the fixed-point format. The fixed-point accuracy of the real and imaginary input values are Q4.11, meaning that the most significant bit is reserved for the sign, 4 bits represents the integer part, and 11 bits represents the fractional part of the value. Therefore, the input pattern of the decoder consists of 30 values of signed 16 bit values in Q4.11 format. This almost fully utilize the vector calculation unit of the DSP, as only two dummy values are calculated during vectorized processing. From the NN model point of view, one of those two dummy values can be used to store the bias constant of 1, so that it does not need to be fetched from the memory separately. The output layer of the network is matched with the 5 possible output options of the NN based decoder.

Collected data consists of 50000 input data samples with corresponding output labels. The data is used both for model training and validation without any split. 24820 data samples contains labels with the DTX outcome and rest 25180 data samples are divided between four other outcomes.

4.2.2 Model architecture and training

Because there are not existing data about the NN inference execution speed on the CEVA-XC4500, the idea is to create as small NN as possible. When the total

number of the NN model parameters is kept minimum, the number of required operations is also kept minimum leading to faster inference. Different feedforward NN architectures is tested still keeping the DSP architecture in mind. Adding new layers to the network increases processing time, as layer need to be fully inferred, before its output can be used as an input for the next layers. Therefore, the layer-wise inference cannot be parallelized efficiently. Instead, the intra-layer inference can be paralleilized in a way that processing of single neuron inferences are overlapped. Within a layer, single neurons can be processed simultaneously in a VCU and different operations can be pipelined. From that point of view, the number of neurons in a single layer can be flexibly chosen. Adding a neuron to a layer is relatively cheap from the execution speed point of view, compared to adding a completely new layer. However, as CEVA-XC4500 can process 16 atomic operations for 32-bit data at one vectorized SIMD instruction, input of a single neuron should be kept divisible by 16. Otherwise either extra instructions need to be executed during handling the tails of the data, or some of the processing capability is wasted.

Figure 4.3 represent the chosen neural network architecture. It consists of input layer with 31 inputs, one hidden layer with 16 hidden neurons, and an output layer with 5 output neurons. Table 4.4 summarize the architecture of the NN. The total number of weight coefficient is 576, and the total number of neurons is 52. The hidden layer uses ReLU-activation function. The output layer uses softmax-activation function during the model training and argmax-activation function in the inference.

Table 4.4: Summary of the chosen NN dimensions.

	Number of neurons	Number of weights
Input layer	31	-
Hidden layer	16	496
Output layer	5	80
Total	52	576

The model itself is constructed, trained and validated in Jupyter-notebook environment utilizing Python 3.6 with Tensorflow core 2.0 and Keras packages. The NN model from figure 4.3 is constructed using Keras' dense layers with explained neuron configurations. Weight values are initialized with Keras' truncated normal initializer with the standard deviaton of 0.1. During the training, L2-regularization is used to shrink the weight coefficients close to zero. It helps to keep the variance of the weight set smaller, enabling higher fixed point precision in the

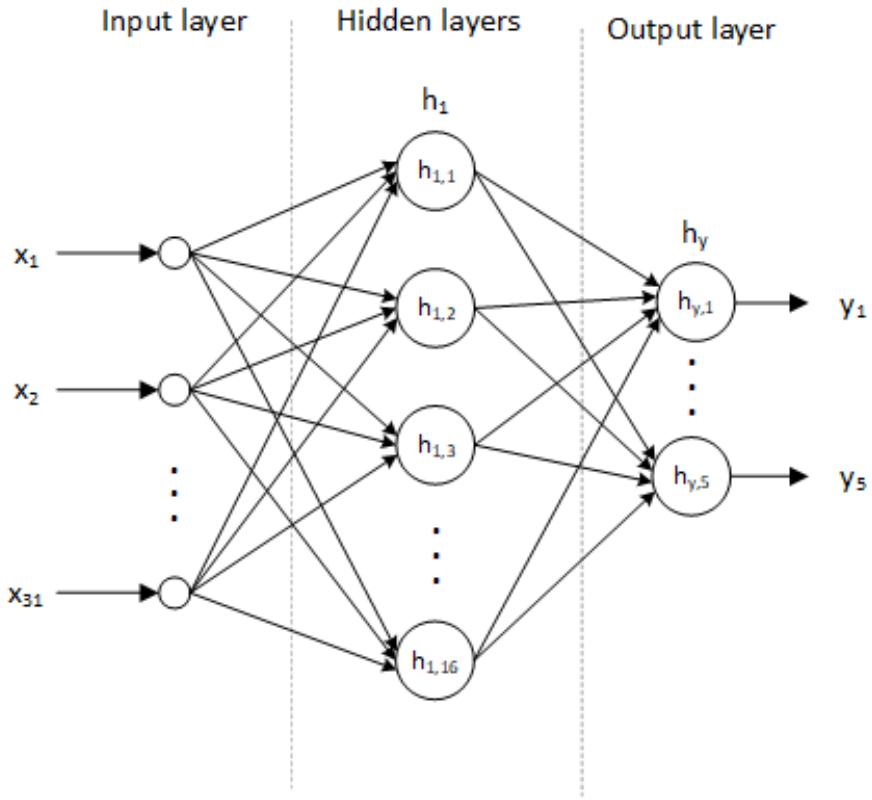

Figure 4.3: Implemented neural network architecture.

DSP implementation. Adam optimizer is used as a backpropagation optimization method for training. Batch size for the training is 128, and 200 epochs are run. One epoch consists of one full training cycle on the training set. Batch size indicates after how many data samples the weight coefficients are updated. The learning rate is also decreased as a function of learning epochs.

Binary cross-entropy is used as a cost function, as it is suitable for classification problems. Traditional BCE is equally weighting all of the outcomes, but in this use case the DTX-class is the most important to be correctly detected. It is especially important to minimize the MDP of the DTX case. For that purpose, a separate gaining matrix G is combined with the cost function to prioritize the correct classification during the training phase. The gaining matrix for the use case is represented in the equation (4.1). $\alpha > 1$ represents a design parameter for the gaining of the cost for MDP of the DTX case (fifth outcome). Dimensions for the gaining matrix and the confusion matrix are same. Equation (4.2) represents the weighted BCE. The traditional BCE from equation (3.12) is weighted with G, resulting higher total cost for misclassified DTX outcome.

$$G(\alpha) = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ \alpha & \alpha & \alpha & \alpha & 1 \end{pmatrix} \tag{4.1}$$

$$C_{G,bce} = -\frac{1}{N} \sum_{i=1}^{N} G(\alpha) * C_{bce}(y_i, p(y_i)) \tag{4.2}$$

4.2.3 Activation functions

ReLU can be efficiently implemented on the CEVA-XC4500 using predicate vectors, explained in the chapter 2.5. Therefore, ReLU is utilized in the hidden layers of the NN. On the output layer, softmax activation is used during the model training, as it provides probabilities for different outcome options. Because the probabilities are out of interest in the final inference implementation, softmax can be replaced with more efficient activation function of argmax. Argmax provides the same classification results as softmax, but without probabilities. Argmax is more efficient to implement in the DSP than the softmax, as it does not include calculation of the exponential functions. Exponential function should be implemented either using pseudo-floating point values or using look-up tables. Floating point based softmax must be implemented on the GPU, which requires more operations than the fixed point and the VCU based argmax. Look-up tables are commonly used method to calculate exponential, but they decrease accuracy and need more memory space.

4.3 DSP implementation

The best way to optimize DSP algorithm is to utilize VCUs. CEVA-XC4500 DSP has a VCU with three independent execution units. The VCU also consists of a predicate mechanism and intra-vector operations. The VCU has a VRF with twelve 256-bit registers. One input register can hold 32 values of 16 bits, so the NN input of 15 complex valued symbols of 16 bits can fit into one input register.

Compiler is also one of the most important methods to optimize a DSP algorithm. When DSP operations and their order are designed carefully, it is much easier for the compiler to utilize pipelining. Also, loops need to be designed in a way they can be unrolled efficiently by the compiler.

The most important metric for the implementation is the DSP cycle count the implementation consumes. The purpose is to create an implementation, which is as fast as possible, but its decoding performance fulfills the 3GPP specification.

The execution speed of the implementation is measured in DSP cycles, and the decoding detection performance is measured with FAR, BER and MDP. The NN model accuracy is measured and validated already in the mode training phase, but the DSP model accuracy need to be validated separately. Rounding the floating point weights to the fixed point weights and also shifting the fixed point precision during multiplications loses accuracy of the inference.

Even thought the execution speed and the decoding accuracy are the only interesting metrics in this implementation, the memory consumption still need to be concerned as a design constraint. The total code size must be less than the internal instruction memory size, and the data must also fit into the internal memory data section. Without the memory size constraint, the model becomes much slower, as the external memory access is very cycle consuming. Also, without the memory size constraint it is not possible to profile the implementation in the CEVA IDE, as the IDE does not simulate external memory.

Implemented NN architecture is represented in figure 4.3 Even though NN architecture is fixed, there are multiple ways how different parts of the implementation can be done differently. In this implementation, software development and optimization is divided into three parts: calculation of induced local fields from the equation (3.3), adding the bias constant and performing neural activations. Two different approaches of the calculation of the induced local field are discussed in section 4.3.2. Comparison between adding the bias in the VCU and in the GCU is explained in the chapter 4.3.3. VCU implementations for ReLU and argmax activations are discussed in section 4.3.4. Fixed point precision of the data is discussed in section 4.3.1.

Tables 4.5 and 4.6 represent explanations for the CEVA intrinsics used in the pseudo-code representations of the implemented algorithms 3-8.

4.3.1 Fixed point format

As the CEVA-XC4500 is optimized for fixed point operations, also the implementation operates purely on fixed point values. The chosen fixed point format depends on the range of the data values. The input symbols of the network come as a result from the different algorithm with the fixed point format of Q4.11. Therefore, the input of the implementation has a fixed point format of Q4.11.

Because L2-regularization is used in the training phase of the network, variance of the weight coefficient remains small. Also, the mean of the weight set is close to zero. Therefore, fixed point format of Q1.15 is enough to cover all possible weight coefficient values. Equation (4.3) represents the range of signed value τ in Qa.b format [13]. According to the equation, the range of the weight coefficients in the implementation in Q1.15 format is $[-2, 1.9999694]$.

Table 4.5: Half of the pseudo-code intrinsics and their explanations used in algorithms 3-8.

Intrinsics	Explanation
load_from_memory(&add)	Loads multiple data from the memory address pointed by &add, and writes it to the vector register file addressed by the return value
VCU_mltply(a, g, K, b, h, L)	Performs dot product for two vectors pointed by a and b. K indicates the word part (LOW, HIGH) used from operand a, and L indicates word part used from operand b. g and h indicates if the first 8 double-words (0) or the last 8 double-words (4) are used from the operands a and b respectively.
VCU_mltply_acc(a, g, K, b, h, L, c)	Performs dot product for two vectors pointed by a and b, and accumulates the result with vector pointed by c. K indicates the word part (LOW, HIGH) used from operand a, and L indicates word part used from operand b. g and h indicates if the first 8 double-words (0) or the last 8 double-words (4) are used from the operands a and b respectively.
VCU_intra_vector_add(a)	Performs intra-vector addition for elements in a vector pointed by a. Operations are performed for both VCUs separately. Results are stored in the first elements in a vector
VCU_to_VCU_move(a)	Performs move operation between two VCUs. Half of the elements from the second VCU are moved into the first VCU.
VCU_add(a , b)	Performs inter-vector addition for vectors pointed by a and b.

Table 4.6: Half of the pseudo-code intrinsics and their explanations used in algorithms 3-8.

Intrinsics	Explanation
VCU_shift(a, k)	Performs shift.operations for elements in a vector pointed by a. Values are shifted by k bits.
VCU_store (&add, a, k)	Stores values in a vector pointed by a into the internal memory address &add. Number of elements to be stored is indicated by value k.
VCU_to_GCU_move(a)	Moves the first element in a vector pointed by a into the GCU.
GCU_add(a, b)	Performs non-vectorized addition operation for values a and b.
VCU_compare_lt(a , k)	Compares if values in a vector pointed by a are lower than value k. The result is a vector predicate, in which 1 indicates the element is lower than k, 0 otherwise.
VCU_fill(a, k, p)	Fills elements in a vector pointed by a with constant k. Predicated vector pointed by p enables or disables atomic operations.
VCU_intra_vector_max(a)	Performs intra-vector max-operation for elements in a vector pointed by an a. Max-operations are executed in pairwise, so that the first and the second elements are compared together, the third and fourth element are compared together and so on. The result from the first comparison is stored as a first element in a return vector, the result from the second comparison is stored as a second elemement and so on.
VCU_compare(a, b)	Compares two vectors pointed by a and b. If an element has the same value in both of the vectors, the corresponding element in the resulting predicate vector has a value of 1, 0 otherwise.

$$-2^a \leq \tau \leq 2^a - 2^{-b} \tag{4.3}$$

4.3.2 Induced local field of a neuron

Induced local field of a neuron is described in the algorithm 3.3. For calculating the induced local field for a layer of neurons, there are two possible parallelization approaches. The first approach is to process one neuron at a time by multiplying all input-weight pairs for a single neuron at once. The second approach is to process all neurons in parallel, but calculating only one input-weight pair per neuron at a time. Pseudocode for the first approach is represented in algorithm 3, and pseudocode for the latter approach is represented in the algorithm 4. Both approaches are evaluated in the chapter 5.

Algorithm 3 Vectorized induced local field, neurons separately

```
1: vec_t inputs = load_from_memory(&input_address)
2: vec_t weight_coeff
3: vec_t acc
4: vec_t temp
5: for every neuron do
6:     weight_coeff = load_from_memory(&weight_address)
7:     acc = VCU_mltply(inputs, 0, LOW, weight_coeff, 0, LOW)
8:     acc = VCU_mltply_acc(inputs, 4, LOW, weight_coeff, 4, HIGH, acc)
9:     acc = VCU_intra_vector_add(acc)
10:    temp = VCU_to_VCU_move(acc)
11:    acc = VCU_add(acc, temp)
12:    acc = VCU_shift(acc, 16)
13:    VCU_store(&result_addr, acc, 1)
```

The first parallelization approach

The first approach in the algorithm 3 is based on the idea that input values, which are common for all neurons within a layer, are loaded from the internal memory to the VRF once, and then reused for every neuron separately. In this way, only neuron specific weight coefficients are loaded from the memory while processing the layer within a loop. As there are 30 inputs of 16 bits in the hidden layer and the weight coefficients are also 16 bits, two vectorized multiplication operations are needed in the CEVA-XC4500 in order to process all input-weight pairs for a single neuron. As input-weight multiplication results are scattered within a vector, intra-vector addition is needed to sum up the single pairs. Intra-vector operation

Algorithm 4 Vectorized induced local field, neurons in parallel

```
1: vec_t inputs
2: vec_t weight_coeff
3: vec_t acc
4: for 8 iterations do
5:     inputs = load_from_memory(&input_address)
6:     weight_coeff = load_from_memory(&weight_address)
7:     acc = VCU_mltply_acc(inputs, 0, LOW, weight_coeff, 0, LOW)
8:     acc = VCU_mltply_acc(inputs, 0, HIGH, weight_coeff, 4, LOW)
9:     weight_coeff = load_from_memory(&weight_address)
10:    acc = VCU_mltply_acc(inputs, 4, LOW, weight_coeff, 0, LOW)
11:    acc = VCU_mltply_acc(inputs, 4, HIGH, weight_coeff, 4, LOW)
12: acc = VCU_shift(acc, 16)
```

operates VCUs separately. As one vectorized instruction is executed in two VCUs, the other intra-vector summation result need to be first moved from one VCU to the other, and then summed together with the result from the first VCU. Finally this summation, the induced local field of a neuron, is shifted back to the 16 bit precision and stored into the internal memory. This process is done for all neurons within a layer. After all neurons are processed, those induced local fields can be loaded back to the VCU register for the activation.

In the algorithm 3, the variable *input* represents the vector register for vectorized processing of the input values. The variable *input_address* represents the internal memory address where the input values are stored. The variable $weight\_coeff$ represents a vector register for vectorized processing of the weight coefficients. The variable *acc* represents a vector register for temporary storing multiplied input-weight-pairs. The variable *temp* is a temporary variable for storing elements from the second VCU, so that they can be added together with the elements in the first VCU. The variable *result_addr* represents the internal memory address where the resulting induced local field value is stored.

In the algorithm 3, for loop is processed 16 times for the hidden layer, and 5 times for the output layer, according to the layer sizes. When loading weight coefficient from the memory in the line 6, the variable *weight_address* is post-inceremented by the intrinsics itself, so that in the next iteration the address is automatically pointing to the next weight set to be loaded. Also when storing the resulting values, the variable *result_addr* is post-incremented by the intrinsics itself. Weight coefficients are represented as 16-bit values, but they are loaded into the VCUs as 32-bit double-word format. Therefore, different word parts, representing different weight values, can be accessed by using LOW or HIGH -levers

in the intrinsic call.

The second parallelization approach

The second approach in the algorithm 4 is based on the idea the input vector is not static, even though the input values are static. All of the neurons on a layer are processed simultaneously, so that one input-weight-pair per neuron is calculated per operation, but for every neuron on a layer. When this process is iterated 32 times for a hidden layer, all of the input-weight-pairs are processed for every neuron on the layer. For every iteration, new vector of inputs and weight coefficients are loaded from the memory to the vector registers. Figure 4.4 illustrates how input vector is processed and loaded from the internal memory in a way that every data element from the input values are placed to every atomic position of the input vector register once. This way, every neuron of the layer are fed with every input value once.

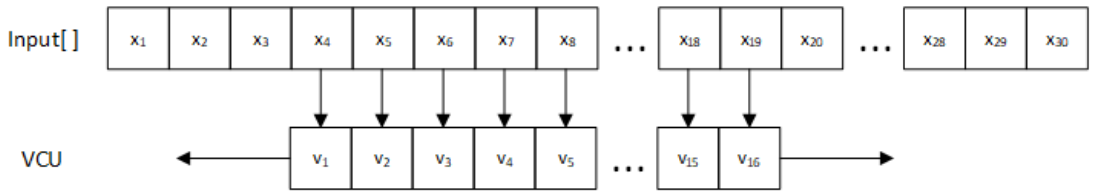

Figure 4.4: Representation how input values are processed for the algorithm 4.

If input values are stored twice to the internal memory in a concatenated manner, the input values can be loaded to the input vector register in different order in every iteration just by modifying the memory address to load the values from. For example, in the first iteration, the memory address points to the value x_1, and values loaded from the memory to the input vector are x_1 - x_{30}. When starting address is increased by the size of one input value, the starting address points to the input value x_2. In the next iteration, the values loaded from the memory are x_2 - x_{30}, and the value x_1 will be the last element of the input vector register. This happens, because the input values were stored twice in the memory.

In the algorithm 4, the variable *inputs* represent the input vector register to store the input values for vectorized processing. The variable $weight\_coeff$ represents the vector register for storing the weight coefficient values for vectorized processing. The variable *acc* represents a vector register to store accumulated and multiplied input-weight-pairs. The variable $input\_address$ points to the internal memory address to load the input values from. The address is post-incremented in the line 5 by the intrinsic, so it always points to the correct address. The variable $weight\_address$ represents the internal memory address to load the

weight coefficient values from, and it is also post-incremented in the lines 6 and 9 by the intrinsic.

With levers g and h described in the Table 4.5 for the $VCU\_multply\_acc$, the input vector can be used four different ways per one for-loop-iteration. Therefore, the input vector need to be loaded from the memory only 8 times in order to use all of the 30 inputs and one bias in every atomic position in the input vector register. However, the output layer processing also needs 8 iterations, even though the layer size is smaller. Otherwise all of the 16 input values are not placed into every atomic position in the input vector register. The g and h levers give the possibility to choose the offset for the first double-word value to be used from the VCU, and this way the first 16 values or latter 16 values can be used. Also, it can be chosen if the most significant (HIGH) or less significant (LOW) part of the double-word is used first.

Even though the input vector can be utilized in four different ways, the result from the MAC-intrinsic can be stored only two different ways. The result value can be stored into the HIGH or LOW part of the resulting double-word. Because one weight vector can contain only two set of 16-bit weights, weight coefficient vector need to be loaded twice in one loop iteration.

The second parallelization approach consumes more memory and does more memory operations than the first parallelization approach, but most of the operations are efficient hardware-based MAC-operations. After 8 iterations, accumulator contains the induced local fields of every neuron in the layer. Compared to the first parallelization approach represented in the algorithm 3, all intra-vector operations, moves from VCU to VCU, and temporary storage of the induced local field values are removed. It is much easier for the compiler to pipeline and to optimize operations, as the main processing contains only MAC-operations and memory accesses.

4.3.3 Bias

As discussed in chapter 3.1.2, the bias constant can be embedded into weight coefficient vector by adding additional constant to the input vector. This is useful feature also for this use case. The input has a length of 30, and there are 2 dummy input values available in the input vector register. One of those can be utilized as bias input without extra computational cost.

Algorithm 5 represents pseudocode of the CEVA-XC4500 implementation of adding the bias separately in a GCU. Algorithm 6 represents pseudocode of adding the bias during the vectorized processing in the VCU. The variable $input\_address$ represents a pointer to the internal memory address for the input values. The variable $weight\_address$ represents a pointer to the internal memory address for the weight coefficient values. In the algorithm 5, the variable $bias\_address$

represents a pointer to the address containing the bias constant. In the algorithm 6, the bias is included into the content pointed by the *input_address* and the *weight_address*. Those memory contents are loaded into the internal memory before any DSP processing. Therefore, no extra processing steps are needed to add the bias to the inference. In algorithm 5 instead, the bias is located in completely different memory location than the weight coefficients. For that reason, the bias needs to be loaded separately from the memory. Because the result from the intra-vector addition in the line 4 is stored within vector registers, the result needs to be moved from the VCU to the GCU before the bias can be added to it.

Algorithm 5 Adding bias constant in the GCU

```
1: vec_t input_features = load_from_memory(&input_address)
2: vec_t weight_coeff = load_from_memory(&weight_address)
3: vec_t product = VCU_multiply(input_features, weight_coeff)
4: vec_t sum = VCU_intravector_multiply(product)
5:
6: int sum = VCU_to_GCU_move(sum)
7: int bias_coeff =load_from_memory(&bias_address)
8: int output = GCU_add(sum, bias_coeff)
9:
10: return output
```

Algorithm 6 Adding bias constant in the VCU

```
1: vec_t input_features = load_from_memory(&input_address)
2: vec_t weight_coeff = load_from_memory(&weight_address)
3: vec_t product = VCU_multiply(input_features, weight_coeff)
4: vec_t sum = VCU_intravector_multiply(product)
5:
6: return sum
```

4.3.4 Activation functions

As discussed earlier in this chapter, ReLU is chosen as an activation function for the hidden layer of the chosen NN architecture. The output layer utilizes softmax activation function in the NN model, but it is replaced with argmax activation for the DSP inference. Outcome probabilities are out of interest in the real-time inference and only the outcome with the highest probability is interesting. Therefore, the computationally more expensive softmax activation can be replaced with the argmax activation.

ReLU implementation

Algorithm 7 represents pseudocode for vectorized ReLU implementation for the hidden layer. ReLU is a activation function for a single neuron, but all of the activations can be computed in parallel for a layer. In order to avoid branching, vector predicate method explained in the chapter 2.5 is utilized to replace all negative induced local field values with zero. The variable *input* represents a vector register containing induced local fields of a layer. The variable *negative_values* is a vector predicate containing information about all of the negative elements in the *input* vector after the compare-intrinsic is processed. The variable *result* represent a vector register containing ReLU-activated inputs.

Algorithm 7 Vectorized ReLU for CEVA-XC4500

```
1: %Process induced local field for all neurons in a layer
2: % vector register "input" contains all of the induced local fields
3:
4: vpred_t negative_values = VCU_compare_lt(input, 0)
5: vec_t result = VCU_fill(input, 0, negative_values)
6:
7: return result
```

In the algorithm 7, the precondition is that all of the induced local fields for the layer are processed and stored in the vector register file *input*. In this use case, they are stored as 32-bit values. VCU's comparison intrinsic, $VCU\_compare\_lt()$, is utilized to compare content of the *input* with constant zero, which is the threshold value in the ReLU. The comparison result is stored into vector predicate register containing binary 1 for the elements in the *input* vector having value less than zero, and binary 0 otherwise. The VCU intrinsic $VCU\_fill()$ replaces values from *input* with zeros, according to the predicate vector *negative_values*. The fill operation is executed for every atomic value in the *input*, but the result is stored to the vector register *result* only for the values marked by 1 in the predicate vector.

Argmax implementation

Algorithm 8 represents pseudocode for vectorised argmax implementation. Similary than in algorithm 7, the vector register file *input* contains the induced local fields for all of the neurons in a layer. In this use case, the *input* contains 16-bit values. The variable *output* represents the vector register for storing maximum values returned by the intrinsic $VCU\_intravector\_max()$. As explained in the Table 4.6, the intrinsic $VCU\_intravector\_max()$ performs intra-vector maximum operation between 16-bit wide sources that are located in the same vector. Figure 4.5

represents the VCU processing of intra-vector max operation. Because the intrinsic compares elements in a pairwise manner, the intrinsic need to be called four times in order to get the maximum of all values in the register. After four iterations, the vector *output* in algorithm 8 contains the maximum value in every element of the vector register. This vector can be compared with the original input vector *input* in order to find the index of the maximum.

Algorithm 8 Vectorized argmax for CEVA-XC4500

```
%Process induced local field for all neurons in a layer
% vector register "input" contains all of the induced local fields

vec_t output = VCU_intravector_max(input)
output = VCU_intravector_max(output)
output = VCU_intravector_max(output)
output = VCU_intravector_max(output)

vpred_t index = VCU_compare(input, output)
int result = VCU_to_GCU_move(index)

return result
```

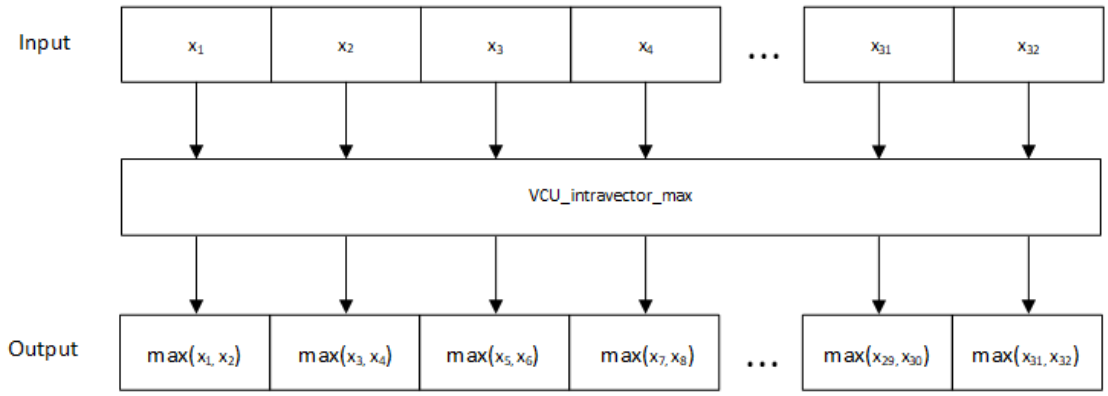

Figure 4.5: Intra-vector maximum operation in a VCU.

Chapter 5

Evaluation

The objective of this thesis was to implement optimal neural network inference for CEVA-XC4500 digital signal processor under hard real-time constraints. The use case, neural network model and the model's DSP implementation was described in chapter 4. In this chapter, the trained NN model performance and different DSP implementations are evaluated. At first, the NN model evaluation metrics and the evaluation results are discussed. After that, the DSP implementation metrics and the implementation results are evaluated.

As absolute cycle counts are confidential property of Nokia, the measured cycle counts in section 5.2.2 are represented as a relative count to the optimized, conventional stochastic model based decoder with compiler optimization level -O3. However, the purpose was to study if a neural network based model could improve the system performance from an execution speed point of view without compromising accuracy requirements. For that purpose, the relative cycle count is actually an even more interesting metric than absolute cycle values.

5.1 Neural network model

5.1.1 Evaluation metrics

The neural network model described in chapter 4 is evaluated by different metrics. The NN model is evaluated based on the accuracy, precision, recall and f1-score of individual classes or the whole dataset. The NN model is compared to the convectional decoder by calculating the misdetection probability (MDP), false-alarm rate (FAR) and bit-error rate (BER). As described in chapter 4.1, classes 1-4 represent different decoder outcomes based on the combination of the two inputs and class 5 represents the DTX outcome.

Accuracy of the model describes how many samples from the data set are

classified correctly. It is a fraction of the correctly classified samples and the total size of the test set. Precision of the model describes the fraction of correctly classified samples of a given class and all samples that were predicted to belong to the same class. Recall describes the fraction of correctly classified samples of a given class and the number of samples that actually belong to that class. F1-score is a weighted combination of recall and precision according to equation (5.1).

$$f1 - score = 2 * \frac{precision * recall}{precision + recall} \tag{5.1}$$

MDP refers to the probability that the decoder falsely decodes a class 1-4 sample as a class 5 sample. FAR is the probability that the decoder falsely decodes a class 5 sample as a class 1-4 sample. BER refers to a probability that the sample is falsely decoded as a class 1-4 sample. In the 3GPP specification, all of those probabilities are required to be less than 10^{-2} [52]. The table 5.1 represents MDP, FAR and BER for the NN based decoder.

Table 5.1: MDP, FAR and BER of the NN based decoder.

	NN based decoder	3GPP specification requirement [52]
MDP	0.58 %	1.0%
FAR	0.17 %	1.0%
BER	0.17 %	1.0%

A confusion matrix is a table that is used to describe the performance of the NN model. A confusion matrix is a summary of prediction results on a classification problem. The number of correct and incorrect predictions are summarized with count values and broken down by each class. Table 5.2 represents the confusion matrix of the implemented NN model. Each column represents a predicted class from 1-5, and each column represents the true class. The diagonal in the confusion matrix represents correctly classified samples.

5.1.2 Evaluation

Table 5.2 represents the confusion matrix of the implemented NN model with the test data set of 50 000 samples. Rows in the confusion matrix represent the true labels of the data points, and columns represent predicted outcomes by the NN model. Each cell contains cumulative count for the outcome. The diagonal of the confusion matrix represent correctly classified samples. Other evaluation metrics can be calculated using the values in the confusion matrix.

Table 5.2: Confusion matrix with the validation set.

		Predicted lables by the NN decoder				
		1	2	3	4	5
True labels	1	6252	0	0	0	43
	2	0	6260	0	0	35
	3	0	0	6261	0	34
	4	0	0	0	6264	31
	5	15	8	9	9	24779

Table 5.3 represent precision, recall and f1-score of the NN model with the test data. Values are calculated from the confusion matrix. Table 5.1 represents MDP, FAR and BER of the NN model tested with the test data set. Those metrics are also calculated using the confusion matrix.

Table 5.3: Precision, recall and f1-score of the NN.

	Precision	Recall	f1-score
1	1.00	0.99	1.00
2	1.00	0.99	1.00
3	1.00	0.99	1.00
4	1.00	1.00	1.00
5	0.99	1.00	1.00

Based on the NN decoder performance represented in the confusion matrix in the Table 5.2, the overall accuracy of the model is 99.63%. As the Table 5.1 illustrates, MDP of the NN based decoder is 0.58%, the FAR is 0.17%, and the BER is 0.17%. Those three metrics are less than 1% specified in the 3GPP-specification [52]. Therefore, the NN based decoder works as well as conventional decoder from the decoding point of view, fulfilling all of the specification requirements.

As the Table 5.3 represents, all of the model metrics precision, recall and f1-score are at least 99% for all of the individual classes. Therefore, the overall performance of the NN is excellent. As the confusion matrix in the Table 5.2 illustrates, the decoder works perfectly with the classes 1-4, if it can separate the sample from the class 5. The class 5 sample seems to be the only difficult sample to decode. However, the overall performance of the NN based decoder is great.

5.2 DSP implementations

Different DSP implementations A-G and their differences are explained in the following subsections. Implementation A is the reference implementation, containing optimized stochastic model based implementation with compiler optimization level -O3. Implementations B-G are profiled without compiler optimizations (-O0) and with the compiler optimization level -O3.

Implementation A

Implementation A is the reference implementation, containing stochastic model based decoder (and DTX-decision). This implementation is not related to the machine learning. This implementation is fully optimized, and it is compiled with the optimization level -O3. This implementation is currently used in real Nokia baseband products.

Implementation B

Implementation B is another reference implementation. This implementation is a NN inference, but it is coded purely on C language without any manual optimizations or DSP related intrinsics. It is totally reusable and scalable for different applications. It can be compiled and executed on any computer without code modifications.

Implementation C

Implementation C is a NN inference with some manual optimizations. It uses the approach described in the algorithm 3, in which neurons in a layer are inferred one at a time. The bias is excluded from the input as described in the algorithm 5. In this implementation, the induced local field calculations are performed in a VCU, but activation functions follow GCU-based calculus.

Implementation D

Implementation D is similar to the implementation C, except that the activation functions are also calculated in the VCU as described in the algorithms 7 and 8. The bias is still excluded from the input according to the algorithm 5, and the induced local field is calculated in a neuron after neuron basis as described in the algorithm 3.

Implementation E

Implementation E is similar to the implementation D, except that the bias is added to the input. Therefore, the bias is added to the induced local field calculus in the VCU as described in the algorithm 6. The induced local field is calculated in neuron after neuron basis according to the algorithm 3, and the activations are also performed in the VCU as described in the algorithms 7 and 8.

Implementation F

Implementation F follows the second approach for calculating the induced local fields for a layer, as described in the algorithm 4. The induced local fields for all neurons in a layer are calculated in parallel. However, only the hidden layer is calculated in this manner in this implementation. The induced local fields of the output layer are calculated following the first approach described in the algorithm 3. In this implementation, the bias is included in the input and added to the induced local field according to the algorithm 6. Activation functions are calculated in the VCU as described in the algorithms 7 and 8.

Implementation G

Implementattion G is similar to the implementation F, except that also the hidden layer is calculated following the second approach described in the algorithm 4. Therefore, both layers are inferred following the same approach. The bias is still part of the input and calculated according to the algorithm 6. The activations are calculated in the VCU as described in the algorithms 7 and 8. In this implementation, all massive operations are performed in the VCU.

5.2.1 Evaluation metrics

DSP implementations are evaluated based on the execution speed. As described in the chapter 2.3.1, when the main clock frequency of the system is constant, the execution speed is proportional to the cycle count the implementation consumes. Because the CEVA-XC4500 has a constant clock frequency, the execution speed can be measured in cycle counts.

In this work, two methods are used to measure cycle counts. The first method is the profiling tool found in the CEVA IDE, and the second method is the Lauterbach profiling tool.

The profiling tool in CEVA IDE calculates the cycles offline based on the compiler output file. The cycle measurement is accurate, if the code and data are small enough to fit into the internal memory of the DSP. All of the implementations A-G fit into the internal memory of the CEVA-XC4500, so the profiling results are

valid. All the results represented in the figures 5.1 and 5.2 are profiled with CEVA IDE profiling tool.

The other method, the Lauterbach profiling tool, is based on the product called Lauterbach PowerTrace module. It is a non-intrusive on-chip tracing hardware attached to the real baseband unit and the DSP core via PCI express bus. It captures program trace from the target harware and is totally cycle accurate. This profiling method is utilized with implementations A, B and C in order to verify the results from the CEVA IDE profiling tool. Profiling results match, so the CEVA IDE profiling results can be used as an accurate information.

5.2.2 Evaluation

Figures 5.1 and 5.2 represent relative cycle counts for implementations A-G with compiler optimization levels -O0 and -O3 respectively. The implementation A is an exception, as it is compiled with the compiler optimization level -O3 in both figures. Therefore, it can be used as a global reference for all implementations with different optimization levels. Details for different implementations are explained in section 5.2. Key differences between compiler optimization levels -O0 and -O3 are that the -O3 does loop unrolling, pipelining and combines instructions if possible.

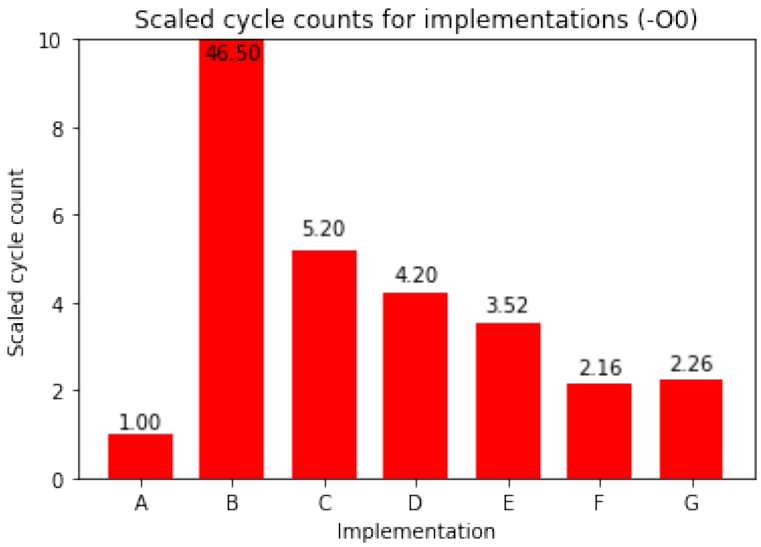

Figure 5.1: Relative cycle counts for different DSP implementations without compile optimizations. Conventional stochastic model based decoder implementation with compiler optimization level -O3 as a reference (A).

As figure 5.1 illustrates, the implementation B is 4550% slower than the reference implementation A. This implementation does not utilize the VCUs at all, nor do any

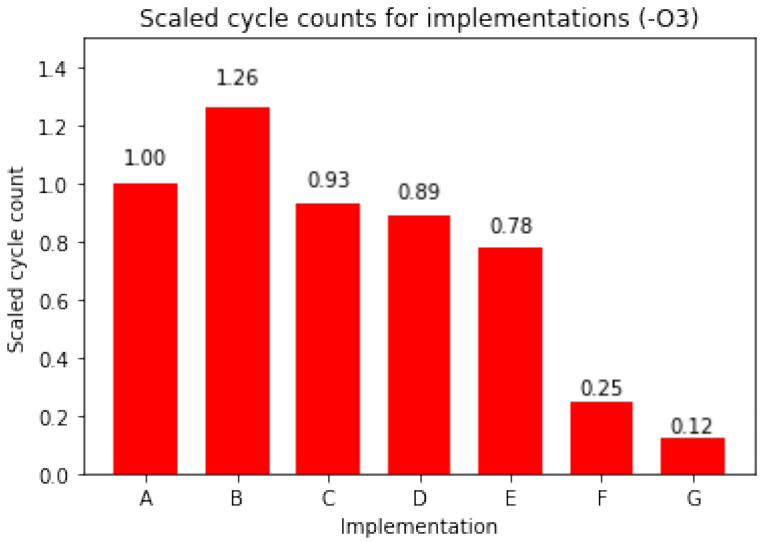

Figure 5.2: Relative cycle counts for different DSP implementations with compiler optimization level -O3. Conventional stochastic model based decoder implementation as a reference (A).

software pipelining. This implementation does the processing completely unparallel. It probably performs similarly in a DSP and in any GPP. However, when compiler is allowed to optimize it, pipelining, loop unrolling and software inlining optimize it a lot, leaving it still to be 26% slower than the conventional reference implementation A, as figure 5.2 illustrates. Therefore, reusable neural network model with only compiler optimizations is not as fast as optimized conventional decoder. This illustrates the need for manual optimizations.

Difference between implementations B and C in figure 5.1 is remarkable. Just by computing induced local fields in the VCU reduces the cycle count by almost 89%, reducing scaled cycle count from 46.5 to 5.2. VCU utilization adds parallel processing and makes single multiplication and addition operations much faster. However, the implementation B is still 420% slower than conventional decoder implementation A. With the compiler optimizations in figure 5.2, VCU utilizations enables even more pipelining possibilities. Also the code structure is easier to unroll. With the compiler optimizations, the implementation C consumes 7% less cycles than the conventional decoder.

Difference in the cycle counts between implementations C and D in a figure 5.1 is about 19%, making the implementation D 320% slower than the implementation A. As figure 5.2 illustrates, the implementation D consumes 11% less cycles than the implementation A with compiler optimizations. The gain comes from the VCU utilization in the output layer activation. However, the output layer activation

is performed only once during the inference, and the amount of operations in the argmax-activation is very small compared to the amount of operations when inferring induced local field for total of 576 input-weight-pairs, as counted in the Table 4.4. Also, the argmax-activation includes mostly intra-vector operations, which are generally slower compared to the inter-vector calculations. Therefore, the gain in the compiler optimized version is only 4% in cycle counts.

Implementation E includes predicated ReLU-activation for the hidden layer. The cycle reduction compared to the implementation D is about 16% without compiler optimizations, as illustrated in the figure 5.1. This makes the implementation 252% slower than implementation A. In the compiler optimized version in the figure 5.2, the cycle reduction is about 8% compared to the implementation D, reducing the cycle count 22% compared to the conventional implementation A. Vector predicate mechanisms reduces the amount of processing, making the implementation more parallel. The cycle count result in this case also depends on the number of negative outputs in the induced local field inferences, as by default, the execution expects positive comparison result. The cycle penalty is added only when this expectation is not met.

Implementations F and G utilize the second approach for the calculation of the induced local fields inference, as described in the algorithm 4. This optimizes the implementation a lot. The total number of multiplications remains same, but all of them are executed as hardware-based MAC-operations, so the addition comes without extra cycle cost. In the implementations C-E, only half of the multiplications are MAC-operations. There are not intra-vector operations in the F and G, whereas in the implementations C-E, the inference of the every neuron includes intra-vector additions and VCU to VCU moves. The total number of operations is reduced, which itself reduces the cycle count a lot. Inter-vector operations are also easier to pipeline than intra-vector operations. Only addition to the processing is the modification of the input, as described in section 4.3.2. Without compiler optimizations, the implementation F consumes about 37% less cycles than the implementation E, and the implementation G consumes about 36% less cycles than the implementation E. Without compiler optimizations, the implementation F is 116% slower than the reference implementation A, and the implementation G is 126% slower than the implementation A. It is interesting to realize that the implementation F consumes actually 4.6% less cycles than the implementation G without compiler optimizations. This is due to added memory accesses for modifying the input of the output layer. However, when compiler optimization level -O3 is used, as figure 5.2 illustrates, the implementation G consumer 52% less cycles than the implementation F, 84.6% less cycles than the implementation E, and 88% less cycles than conventional decoder implementation A. Pipelining, unrolling, reduction in operations and the maximum VCU utilization

makes the implementation G very speed efficient compared to the conventional method.

5.3 Evaluation summary

The objective of this thesis was to implement optimal neural network inference for CEVA-XC4500 digital signal processor under hard real-time constraints. The use case, neural network model, and model's DSP implementation was described in the chapter 4. In this chapter, NN model and the DSP implementation were evaluated based on different metrics. NN model evaluation was based on the MDP, FAR and BER, defined in the 3GPP specification requirements. The NN model was also evaluated based on accuracy, recall, precision and f1-score. The DSP implementation was evaluated based on the relative cycle count compared to the conventional decoder implementation.

As the results in the chapter 5.1.2 showed, the NN based decoder fulfills all of the 3GPP specification requirements of 1.0% for the MDP, FAR and BER. Also, the overall NN classification performance is excellent, as the total accuracy of the model is 99%, and all of the individual precision, recall and f1-scores for different classes are at least 99% each.

The DSP implementation is also very fast compared to the conventional decoder. As the Table 5.2.2 illustrates, the most optimized implementation option consumes only 12% of the cycle count the conventional implementation consumes, making it more than 8 times faster.

Chapter 6

Conclusions

Machine learning and neural networks in particular are powerful tools for classification analysis, and are used in a wide range of applications. Their potential in physical layer software has also received attention in the field's literature lately. This thesis aimed to create optimized feedforward neural network inference implementation for CEVA-XC4500 digital signal processor. The purpose was to study if neural network based DSP implementation could solve the given decoder use case faster than the conventional stochastic model based decoder implementation. The implementation was based on the NN model, which was created based on the data from the Matlab-simulation model. The model implementation speed performance was compared against conventional stochastic mathematical model based symbol decoder implementation, and different implementation strategies were also compared against each others. The evaluation consisted of classification performance evaluation for the NN model itself, and speed performance measurements for the implementation.

Th results show that even the CEVA compiler itself is capable of doing massive optimizations for the C-based NN source code, if the source code is developed from the DSP optimization point of view. It seems to be significant that the implementation architecture and processing order is chosen correctly. The compiler performs poorly if the code is wholly reusable on any machine, and the processing order is not planned comprehensively. Even though the compiler itself can do subservient optimizations, manual optimizations by the developer are still needed in order to achieve maximum speed performance. Especially, VCU utilization is very important.

The results also show that the implemented NN model is able to decode the given use case in a way that it passes the 3GPP specification performance requirements. The specification performance was passed, even the NN model was designed to be as small as possible due to fact there was no previous knowledge about the speed

performance of a NN implementation in a CEVA-XC4500. The results show that the implemented NN was 88% faster than the conventional decoder, thus the NN model could have been extended. The bigger and especially deeper NN model could have further improved the decoding performance, still being many times faster than the conventional decoder. Implementation in this thesis gives very promising results from the productization point of view. The optimal neural network implementation requires comprehensive knowledge about the target processor. It is important to tailor neural network architecture to be suitable for the target DSP, so that it's VCUs can be fully utilized. Fixed point calculus does not seem to be a problem for a small network, as the cumulative error that is generated during swift-operations is relatively small.

For future research, bigger and deeper neural networks could be researched for the given use case. Also, other types of neural network models, such as CNN and RNN, and their speed performances on a CEVA-XC4500, could be researched. NN inference speed performance could also be evaluated against some ML dedicated hardware-accelerator chip, and they could be also compared from the development and manufacturing cost point of view. Also, multicore DSP inference and its optimal implementation could be considered, as it will enable even bigger NNs.

Bibliography

[1] J. G. Andrews, S. Buzzi, W. Choi, S. V. Hanly, A. Lozano, A. C. Soong, and J. C. Zhang, "What will 5g be?" *IEEE Journal on selected areas in communications*, vol. 32, no. 6, pp. 1065–1082, 2014.

[2] T. Wang, C.-K. Wen, H. Wang, F. Gao, T. Jiang, and S. Jin, "Deep learning for wireless physical layer: Opportunities and challenges," *China Communications*, vol. 14, no. 11, pp. 92–111, 2017.

[3] Z. Qin, H. Ye, G. Y. Li, and B.-H. F. Juang, "Deep learning in physical layer communications," *IEEE Wireless Communications*, vol. 26, no. 2, pp. 93–99, 2019.

[4] R. Oshana, *DSP for Embedded and Real-Time Systems*, 1st ed. Newnes, 2012.

[5] B. D. T. Inc, "Evaluating dsp processor performance," 2000.

[6] A. Bateman and I. Paterson-Stephens, *The DSP Handbook: Algorithms, Applications and Design Techniques*. Prentice Hall, 2002.

[7] S. J. O. Phillip A. Laplante, *Real-time systems design and analysis: tools for the practitioner*, 4th ed. New York: Wiley, 2011.

[8] P. Vernon, "Systems in engineering," *IEE Review*, vol. 35, no. 10, pp. 383–385, Nov 1989.

[9] P. Lapsley and G. Blalock, "How to estimate dsp processor performance," *IEEE Spectrum*, vol. 33, no. 7, pp. 74–78, July 1996.

[10] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[11] C. Inacio and D. Ombres, "The dsp decision: fixed point or floating?" *IEEE Spectrum*, vol. 33, no. 9, pp. 72–74, Sep. 1996.

[12] R. Oshana, *DSP software development techniques for embedded and real-time systems*, 1st ed. Newnes, 2006.

[13] R. Yates, "Fixed-point arithmetic: An introduction," *Digital Signal Labs*, vol. 81, no. 83, p. 198, 2009.

[14] E. Lai, *Practical Digital Signal Processing for Engineers and Technicians*, ser. Electronics & Electrical. Newnes, 2004.

[15] E. A. Lee, "Programmable dsp architectures. i," *IEEE ASSP Magazine*, vol. 5, no. 4, pp. 4–19, 1988.

[16] N. Dahnoun, *Multicore DSP: From Algorithms to Real-time Implementation on the TMS320C66x SoC*. Wiley, 2018.

[17] G. S. Murthy, M. Ravishankar, M. M. Baskaran, and P. Sadayappan, "Optimal loop unrolling for gpgpu programs," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2010, pp. 1–11.

[18] S. Liao, S. Devadas, K. Keutzer, S. Tjiang, and A. Wang, "Code optimization techniques in embedded dsp microprocessors," *Design Automation for Embedded Systems*, vol. 3, no. 1, pp. 59–73, 1998.

[19] C. H. Gebotys and R. J. Gebotys, "Complexities in dsp software compilation: performance, code size, power, retargetability," in *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, vol. 3. IEEE, 1998, pp. 150–156.

[20] *Ceva Compiler optimizations*, Ceva, 2016.

[21] R. E. Kole, "The impact of language extensions on dsp programming," pp. 191–194, April 1996.

[22] M. D. Kleffner, D. L. Jones, J. D. Hiser, P. Kulkarni, J. Parent, S. Hines, D. Whalley, J. W. Davidson, and K. Gallivan, "On the use of compilers in dsp laboratory instruction," in *2006 IEEE International Conference on Acoustics Speech and Signal Processing Proceedings*, vol. 2. IEEE, 2006.

[23] *Ceva Architecture Specification Vol-I*, Ceva, 2016.

[24] *Ceva-XC4500 Methodology for Optimal Vectorization Application Note*, Ceva, 2014.

[25] C. Xiang, "Ee5904r neural networks lecture 1," University Lecture in National University of Singapore, 2017.

[26] S. Shanmuganathan and S. Samarasinghe, *Artificial Neural Network Modelling*, 1st ed. Springer Publishing Company, Incorporated, 2016.

[27] J. J. Buckley and Y. Hayashi, "Fuzzy neural networks: A survey," *Fuzzy Sets and Systems*, vol. 66, no. 1, pp. 1–13, 1994.

[28] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the National Academy of Sciences*, vol. 79, no. 8, pp. 2554–2558, 1982.

[29] T. Kohonen, "Self-organized formation of topologically correct feature maps," *Biological Cybernetics*, vol. 43, no. 1, pp. 59–69, Jan 1982.

[30] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[31] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[32] H. Simon, *Neural networks and learning machines*, 3rd ed. Upper Saddle River, N.J.: Pearson, 2009.

[33] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[34] M. D. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. V. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, and G. E. Hinton, "On rectified linear units for speech processing," in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, May 2013, pp. 3517–3521.

[35] C. Zhang, P. Patras, and H. Haddadi, "Deep learning in mobile and wireless networking: A survey," *IEEE Communications Surveys Tutorials*, pp. 1–1, 2019.

[36] J. Bruck and J. W. Goodman, "A generalized convergence theorem for neural networks," *IEEE Transactions on Information Theory*, vol. 34, no. 5, pp. 1089–1092, Sep. 1988.

[37] N. Farsad and A. Goldsmith, "Neural network detectors for sequence detection in communication systems," *IEEE Transactions on Signal Processing*, 2018.

[38] S. Linnainmaa, "The representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors," *Master's Thesis (in Finnish), Univ. Helsinki*, pp. 6–7, 1970.

[39] D. E. Rumelhart, G. E. Hinton, R. J. Williams *et al.*, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, no. 3, p. 1, 1988.

[40] Y. Qiao, X. Yang, and E. Wu, "The research of bp neural network based on one-hot encoding and principle component analysis in determining the therapeutic effect of diabetes mellitus," in *IOP Conference Series: Earth and Environmental Science*, vol. 267, no. 4. IOP Publishing, 2019, p. 042178.

[41] K. S. Narendra and K. Parthasarathy, "Gradient methods for the optimization of dynamical systems containing neural networks," *IEEE Transactions on Neural networks*, vol. 2, no. 2, pp. 252–262, 1991.

[42] S. Ruder, "An overview of gradient descent optimization algorithms," 2016.

[43] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014.

[44] T. Hesterberg, N. H. Choi, L. Meier, C. Fraley *et al.*, "Least angle and 1 penalized regression: A review," *Statistics Surveys*, vol. 2, pp. 61–93, 2008.

[45] Z. Deng, K.-S. Choi, Y. Jiang, and S. Wang, "Generalized hidden-mapping ridge regression, knowledge-leveraged inductive transfer learning for neural networks, fuzzy systems and kernel methods," *IEEE transactions on cybernetics*, vol. 44, no. 12, pp. 2585–2599, 2014.

[46] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[47] K. Hornik, "Some new results on neural network approximation," *Neural networks*, vol. 6, no. 8, pp. 1069–1072, 1993.

[48] J. X. Chen, "The evolution of computing: Alphago," *Computing in Science & Engineering*, vol. 18, no. 4, p. 4, 2016.

[49] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A survey of deep neural network architectures and their applications," *Neurocomputing*, vol. 234, pp. 11–26, 2017.

[50] T. O'Shea and J. Hoydis, "An introduction to deep learning for the physical layer," *IEEE Transactions on Cognitive Communications and Networking*, vol. 3, no. 4, pp. 563–575, 2017.

[51] J. Sola and J. Sevilla, "Importance of input data normalization for the application of neural networks to complex industrial problems," *IEEE Transactions on nuclear science*, vol. 44, no. 3, pp. 1464–1468, 1997.

[52] 3GPP, "NR; Base Station (BS) radio transmission and reception," 3rd Generation Partnership Project (3GPP), Technical Specification (TS) 38.104, Jun 2019, version 15.6.0.